MORE FROM

NO STARCH PRESS

SCRATCH CODING CARDS

BY NATALIE RUSK

LEARN TO PROGRAM WITH SCRATCH

BY MAJED MARJI

SUPER SCRATCH PROGRAMMING ADVENTURE!

BY THE LEAD PROJECT

CODING IPHONE APPS FOR KIDS

BY GLORIA WINQUIST AND MATT MCCARTHY

PYTHON FOR KIDS

BY JASON R. BRIGGS

LEARN TO PROGRAM WITH MINECRAFT®

BY CRAIG RICHARDSON

THE LEGO® ARCHITECT

BY TOM ALPHIN

THE LEGO® TRAINS BOOK

BY HOLGER MATTHES

THE LEGO® MINDSTORMS® EV3 DISCOVERY BOOK

BY LAURENS VALK

THE LEGO® MINDSTORMS® EV3 IDEA BOOK

BY YOSHIHITO ISOGAWA

ARDUINO PROJECT HANDBOOK, VOLUME 2

BY MARK GEDDES

THE ARDUINO INVENTOR'S GUIDE

BY BRIAN HUANG AND DEREK RUNBERG

THE MANGA GUIDE TO MICROPROCESSORS

BY MICHIO SHIBUYA, TAKASHI TONAGI, AND OFFICE SAWA

75 Cards

Ages 8+

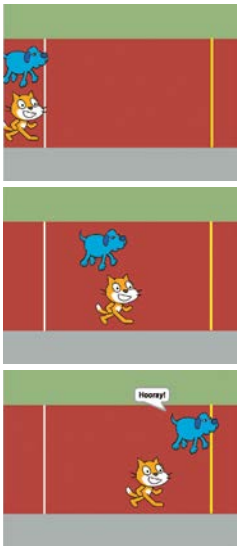
SCRATCH

Coding Cards

Creative Coding Activities for Kids

Choose a Racer

Add another sprite so you can have a race.

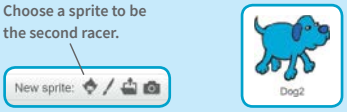


Choose a Racer

scratch.mit.edu/racegame

GET READY

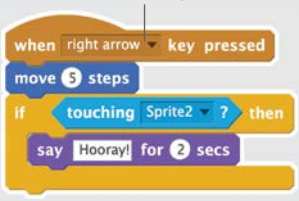
Choose a sprite to be the second racer.



ADD THIS CODE

Drag your sprite to where you want it to start.

Choose **right arrow** or a different key.



TRY IT

Click the green flag to start.

Press the **space** key and the **right arrow** key to make your sprites race.

Make a Conversation

Make your characters talk with each other.

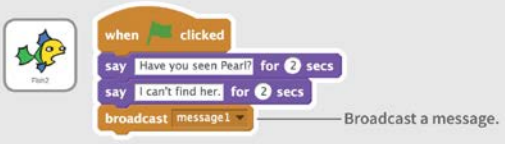
Make a Conversation

scratch.mit.edu/story

GET READY

Choose two characters.

ADD THIS CODE



TRY IT

Click the green flag to start.

TIP

You can click the drop-down menu to add a new message.

Change Backdrops

Click a button to switch backdrops.

Change Backdrops

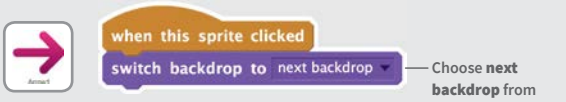
scratch.mit.edu/dressup

GET READY

Choose two backdrops.

Choose a button sprite, like Arrow1.

ADD THIS CODE



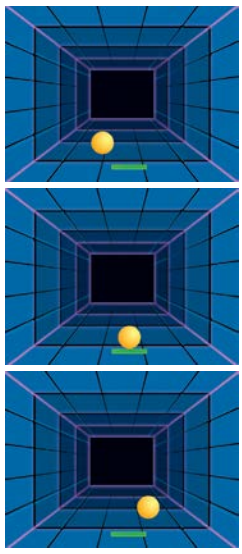
Choose **next backdrop** from the menu.

TRY IT

Click your button to switch backdrops.

Bounce Off the Paddle

Make the ball bounce off the paddle.



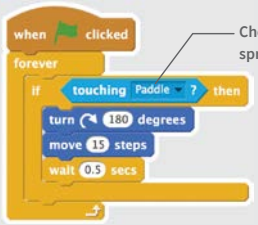
Bounce Off the Paddle

scratch.mit.edu/pong

GET READY

Click to select the **Ball** sprite.

ADD THIS CODE



Choose the **Paddle** sprite from the menu.

TRY IT

Click the green flag to start.

TIP

Insert a **pick random** block to make the ball bounce in different directions.

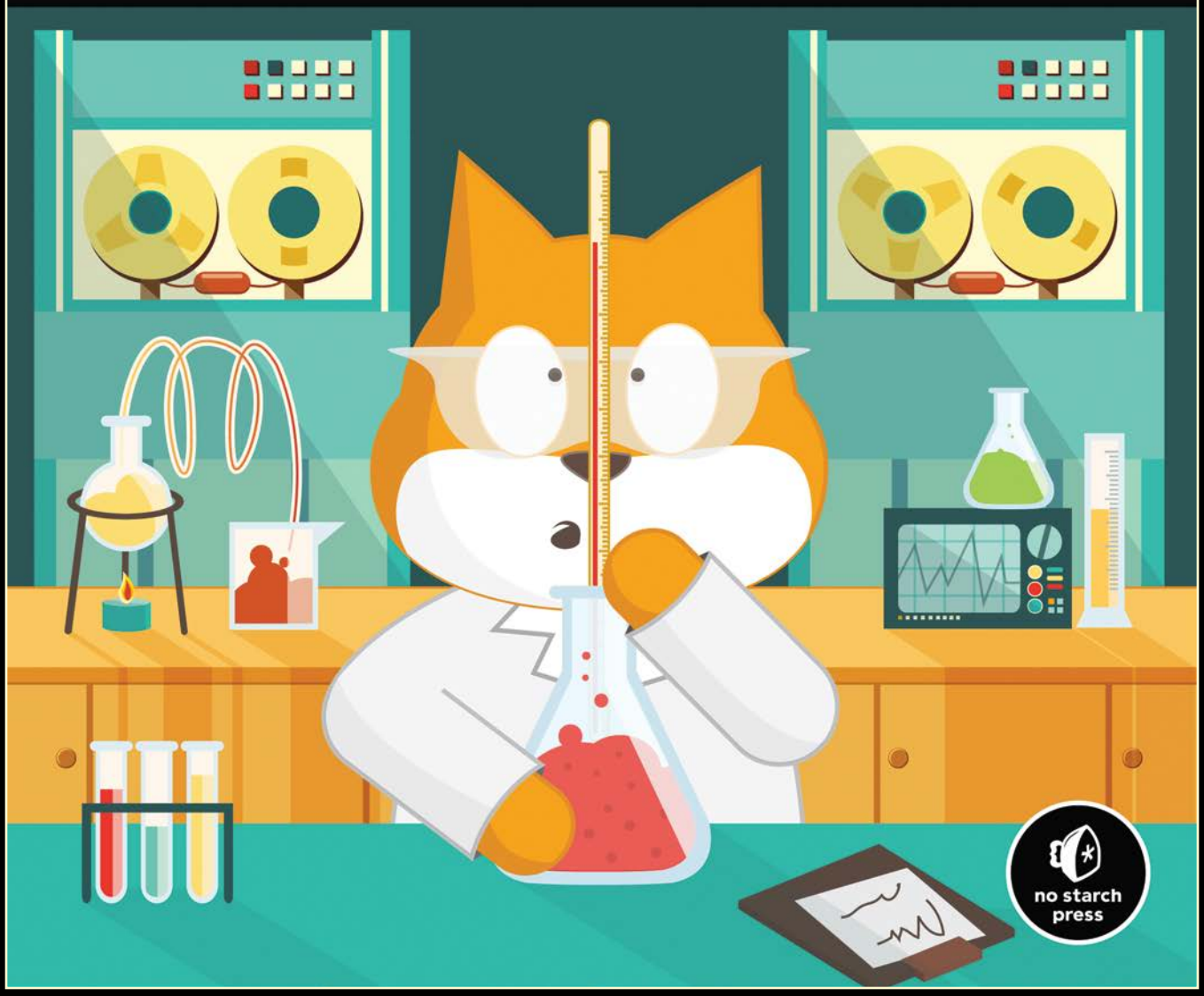
Use numbers around 180.

COVERS SCRATCH 2

LEARN TO PROGRAM WITH SCRATCH

A VISUAL INTRODUCTION TO PROGRAMMING WITH *GAMES*, *ART*, *SCIENCE*, AND *MATH*

MAJED MARJI



no starch press

4

PROCEDURES

This chapter explains how you can take a "divide and conquer" approach to programming. Rather than build your programs as one big piece, you'll be able to write separate procedures that you then put together. Using procedures will make your programs both easier to write and easier to test and debug. In this chapter, you'll learn how to:

- Use message broadcasting to coordinate the behavior of many sprites
- Use message broadcasting to implement procedures
- Use the "build your own block" feature of Scratch 2
- Use structured programming techniques

Most of the applications we've developed so far contain only one sprite, but most applications require multiple sprites that work together. An animated story, for example, might have several characters as well as different backgrounds. We need a way to synchronize the sprites' assigned jobs.

In this chapter, we'll use Scratch's message-broadcasting mechanism to coordinate work among several sprites (this was the only way to implement procedures in the previous version of Scratch). We'll then discuss how to use Scratch 2's "custom blocks" feature to structure large programs as smaller, more manageable pieces called *procedures*. A procedure is a sequence of commands that performs a specific function. For example, we can create procedures that cause sprites to draw shapes, perform complex computations, process user input, sequence musical notes, manage games, and do many other things. Once created, these procedures can serve as building blocks for constructing all sorts of useful applications.

Message Broadcasting and Receiving

So how does the broadcast system in Scratch work in practice? Any sprite can broadcast a message (you can call this message anything you like) using the **broadcast** or **broadcast and wait** blocks (from the *Events* palette) shown in Figure 4-1. This broadcast triggers all scripts in all sprites (including the broadcasting sprite itself) that begin with a matching **when I receive** trigger block. All sprites hear the broadcast, but they'll only act on it if they have a corresponding **when I receive** block.

Figure 4-1: You can use the message-broadcasting and receiving blocks to coordinate the work of multiple sprites.

Consider Figure 4-2. It shows four sprites: starfish, cat, frog, and bat. The starfish broadcasts the jump message, and that broadcast is sent to all sprites, including itself. In response to this message, both the cat and the frog will execute their jump scripts. Notice how each sprite jumps in its own way, executing a different script. The bat also receives the jump message, but it does not act on it because it was not told what to do when it receives this message. The cat in this figure knows how to walk and jump, the frog can only jump, and the bat was taught only to fly.

The **broadcast and wait** command works like the **broadcast** command, but it waits until all message recipients have finished executing their corresponding **when I receive** blocks before continuing.

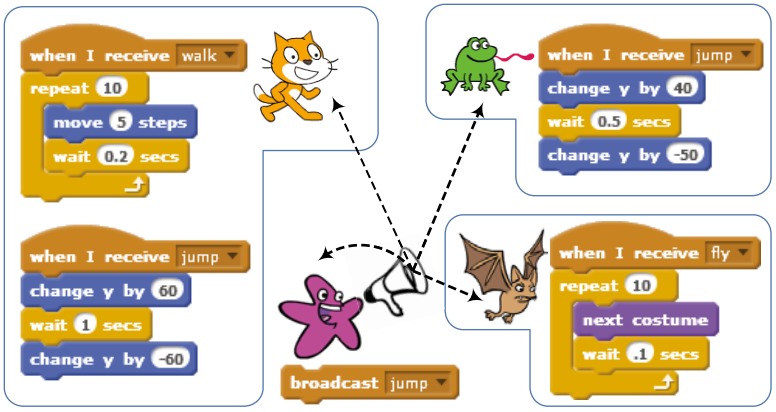


Figure 4-2: A broadcast message is received by all sprites, even by the sprite that sent the broadcast.

Sending and Receiving Broadcasts

SquareApp.sb2

To demonstrate how message broadcasting and receiving work, let's create a simple application that draws randomly colored squares. When the user clicks the left mouse button on the Stage, the Stage will detect this event (using its **when this sprite clicked** block) and broadcast a message that you'll call `Square` (you can choose a different name if you want). When the only sprite in this application receives this message, it will move to the current mouse position and draw a square. Follow these steps to create the application:

1. Start Scratch and then select **New** from the File menu to start a new application. Feel free to change the cat's costume to anything you like.
2. Add the **when I receive** block (from the *Events* palette) to the Scripts Area of the sprite. Click the down arrow in this block and select **new message...** from the drop-down menu. In the dialog that appears, type `Square` and click **OK**. The name of the block should change to **when I receive Square**.
3. Complete the script as shown in Figure 4-3. The sprite first lifts its pen and moves to the current mouse position, indicated by the **mouse x** and **mouse y** blocks (from the *Sensing* palette). It then picks a random pen color, lowers its pen, and draws a square.

The sprite is now ready to handle the `Square` message when it is received. The script in Figure 4-3 can be called a *message handler* since its job is to handle (or process) a broadcast message.

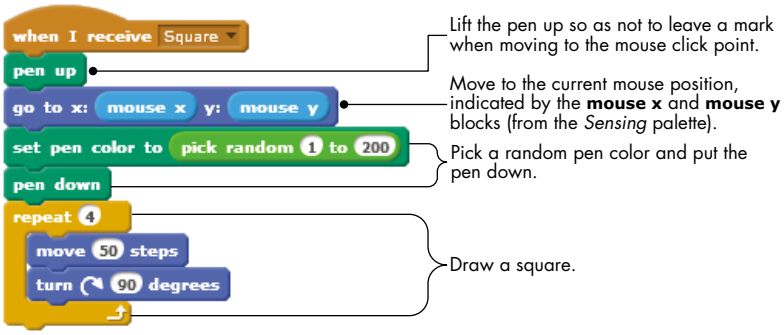


Figure 4-3: The Square *message handler*

Now, let's go to the Stage and add the code to broadcast the Square message in response to a mouse click. Click the Stage in the Sprite List and add the two scripts shown in Figure 4-4. The first script clears any pen marks from the Stage when the green flag is clicked. The second script, which is triggered when the user clicks the mouse on the Stage, uses the **broadcast** block to tell the sprite that it is time to draw.

Figure 4-4: The two scripts for the Stage in the Square *drawing application*

The application is now complete. To test it, just click the mouse on the Stage. It should respond by drawing a square in response to each mouse click.

Message Broadcasting to Coordinate Multiple Sprites

Flowers.sb2

To see multiple sprites respond to the same broadcast message, let's create an application that draws several flowers on the Stage in response to a mouse click. The Flowers application contains five sprites (named Flower1 through Flower5) that are responsible for drawing five flowers on the Stage. Each sprite has its own costume, as shown in Figure 4-5. Note how the background of each costume is transparent. Note also the location of the center of rotation for each costume (marked with the crossed lines).

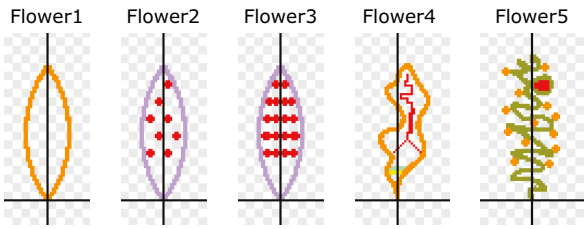


Figure 4-5: Flowers uses these five petal sprites (as shown in the Paint Editor).

When a sprite receives a message to draw its flower, it will stamp multiple rotated copies of its costume on the Stage, as illustrated in Figure 4-6. The figure also shows sample outputs from the flower-drawing script we'll explore next.

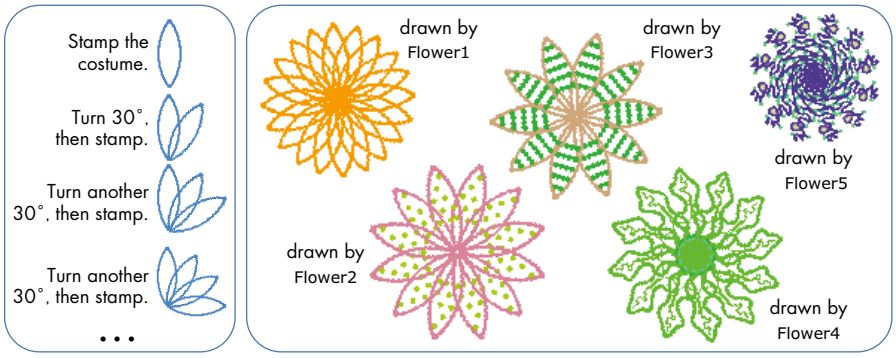


Figure 4-6: The Flowers application's drawing process (left) and some possible flowers (right)

When you click the mouse on the Stage, the Stage detects the mouse click using the **when this sprite clicked** block. In response, it clears its background and broadcasts a message called Draw. All five sprites respond to this message by executing a script similar to the one shown in Figure 4-7.

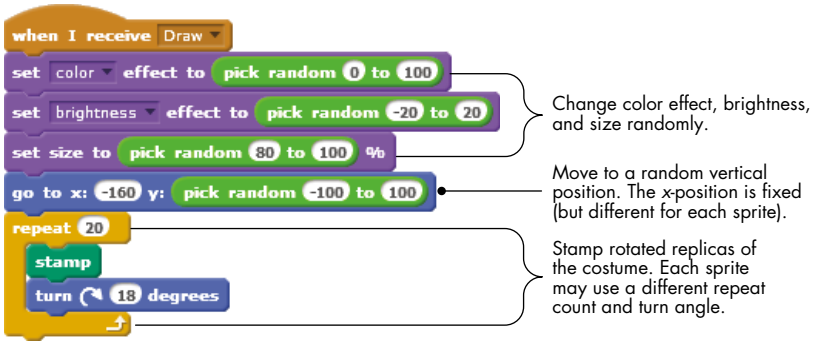


Figure 4-7: The basic script used by each of the five sprites

The script starts by assigning random values to the color effect, brightness effect, and size to change the appearance of the drawn flower. It then moves to a random vertical position on the Stage and draws a flower by stamping rotated replicas of its costume.

Open this application (named *Flowers.sb2*) and run it to see how it works. Despite its simplicity, its output is intriguing. I encourage you to design different costumes to create different types of flowers. Change the costumes' centers to discover even more interesting flower designs.

Now that you understand how message broadcasting and receiving work, we'll move on to introduce structured programming as a way to manage the complexity of large programs.

SUPER SCRATCH PROGRAMMING ADVENTURE!

COVERS VERSION 2

LEARN TO PROGRAM BY MAKING COOL GAMES!

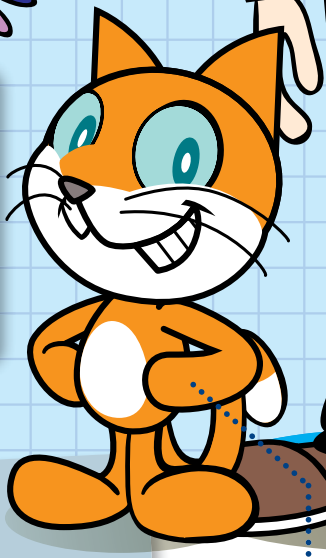
THE LEAD PROJECT

no starch press

MEET THE CAST

Mitch
A computer science student who loves to make cool programs, he's passionate about movies and art, too! Mitch is an all-around good guy.

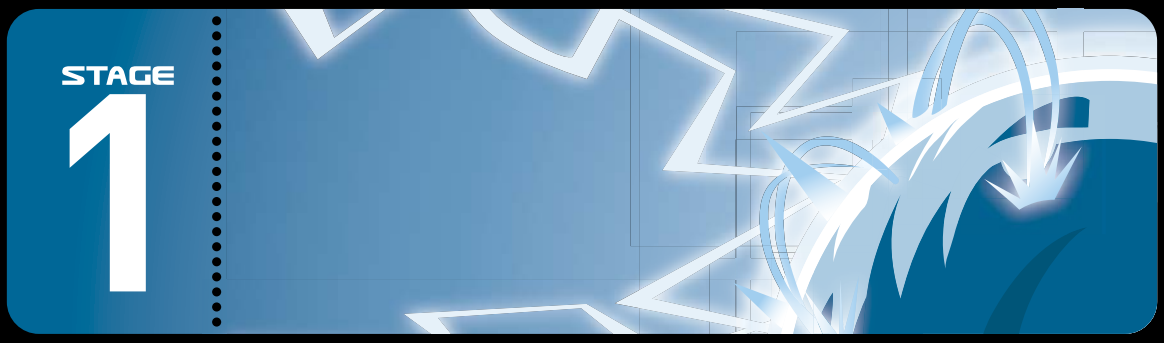
The Cosmic Defenders: Gobo, Fabu, and Pele
The Cosmic Defenders are trans-dimensional space aliens who can travel through space and time. Formally deputized by the Galactic Council, the Cosmic Defender's duty is to maintain the balance of the universe.

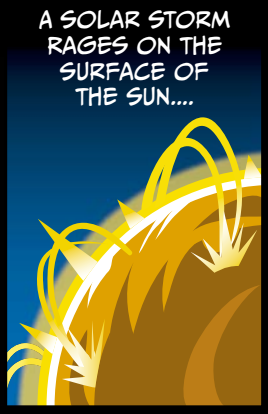


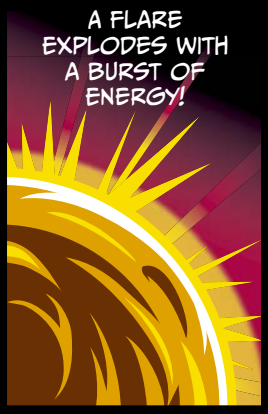
Scratchy
An energetic cat living in cyber-space, Scratchy is exactly what you'd expect from a cat on the Internet. He's quite curious and impulsive.

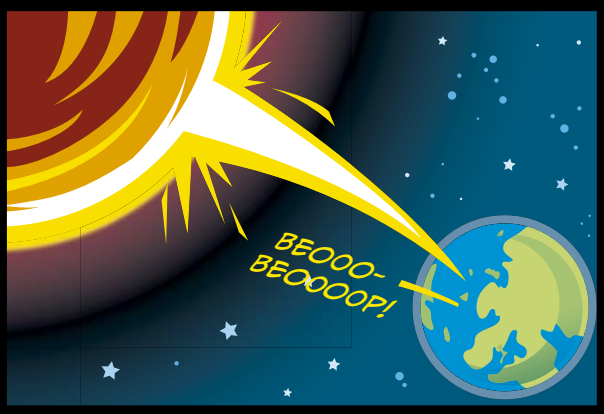
The Dark Wizard
He is a shapeless yet powerful and vengeful spirit, whose origins are unknown. Nothing can stop his ambition of destroying the order of space and time.

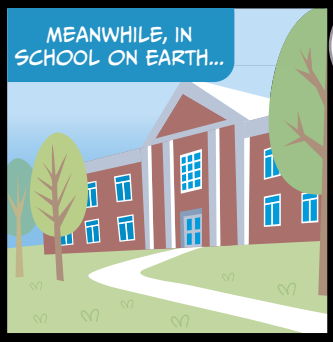
The Dark Minions
These pesky foes are Cosmic Defenders who have fallen to the dark side. They work for the Dark Wizard now.


STAGE
1


A SOLAR STORM RAGES ON THE SURFACE OF THE SUN....

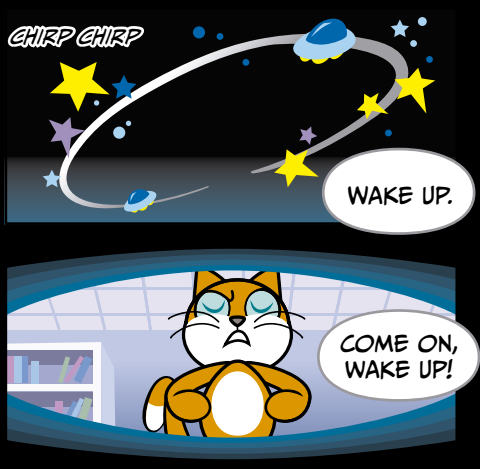

A FLARE EXPLODES WITH A BURST OF ENERGY!

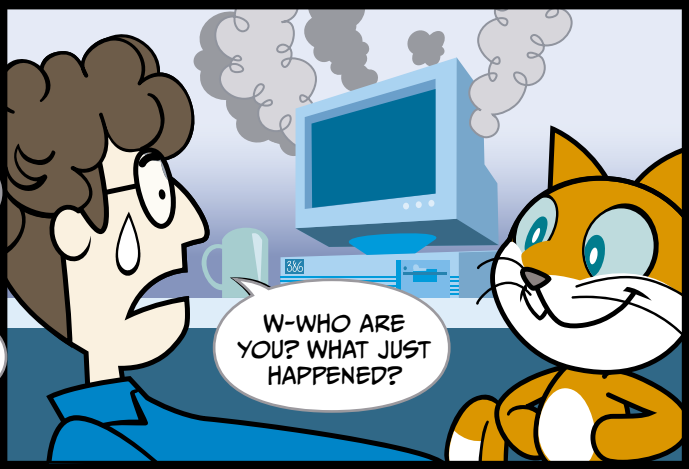

BEOOO-
BEOOOOP!


MEANWHILE, IN SCHOOL ON EARTH...


I SURE WISH PROGRAMMING WERE EASIER...


EARTHQUAKE!
WHOA!


CHIRP CHIRP
WAKE UP.
COME ON, WAKE UP!


W-WHO ARE YOU? WHAT JUST HAPPENED?


MY NAME IS SCRATCHY. I'M FROM CYBERSPACE.
YOU'RE FROM A COMPUTER?!
THAT'S RIGHT. I FOLLOWED THAT FLASH OF LIGHT, AND HERE I AM!
AMAZING! MY NAME IS MITCH. I'M A COMPUTER SCIENCE STUDENT!
HEY MITCH. UM...IS YOUR PLANET ALWAYS SO GRAY?
NO WAY! SOMETHING'S WRONG. LET'S GO CHECK IT OUT!
WHERE IS EVERYBODY?
SOMETHING TERRIBLE IS FORMING IN THE SKY!
OH NO! I CAN'T MOVE MY LEGS!
AHHH! ME NEITHER! HELP!!
STAY CALM! THANK GOODNESS I FOUND YOU!
YOUR BODY IS BEING FROZEN!
TAKE THIS SECRET MANUAL BEFORE THE BLACK TORNADO SWALLOWS YOU! THEN FOLLOW MY INSTRUCTIONS!
LET'S DO THIS!

STAGE 1

BREAKING THE SPELL!

Chapter Focus

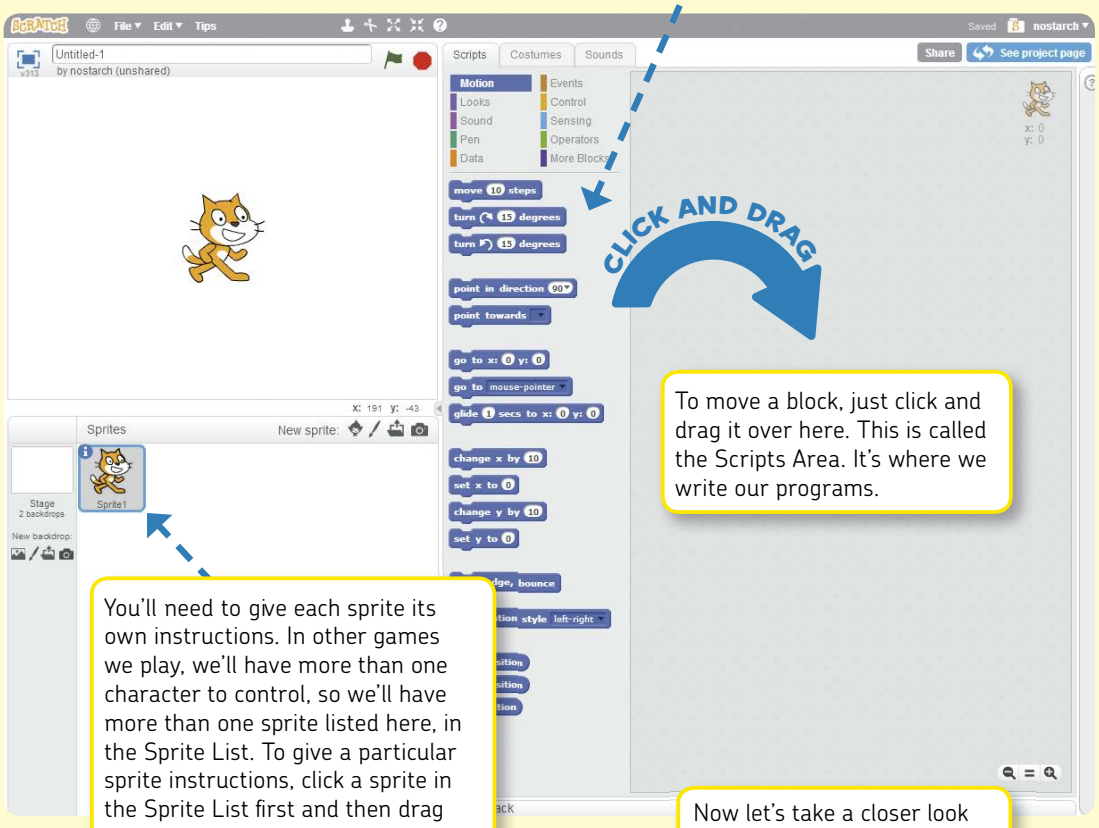
Let's get to know Scratch! We'll also learn about *sprites* and *coordinates*.

The Game

We need to get Scratchy the cat moving again. We'll make him dance across the Stage.

To follow along with the Secret Manual, you first need to open Scratch. Once you **Create** a new project, you'll see Scratchy the cat on a white backdrop. The cat doesn't do anything yet because he doesn't have any programs. Scratch calls Scratchy the cat—and all the other characters and objects we add to a project—a *sprite*. Soon, we'll start giving him directions to move by using the blue blocks in the middle of the screen.

The command blocks you can give a sprite are here. We'll stack these commands together to break the magic spell and get Scratchy back on his feet. The blocks here are all blue, as they're from the **Motion** palette.

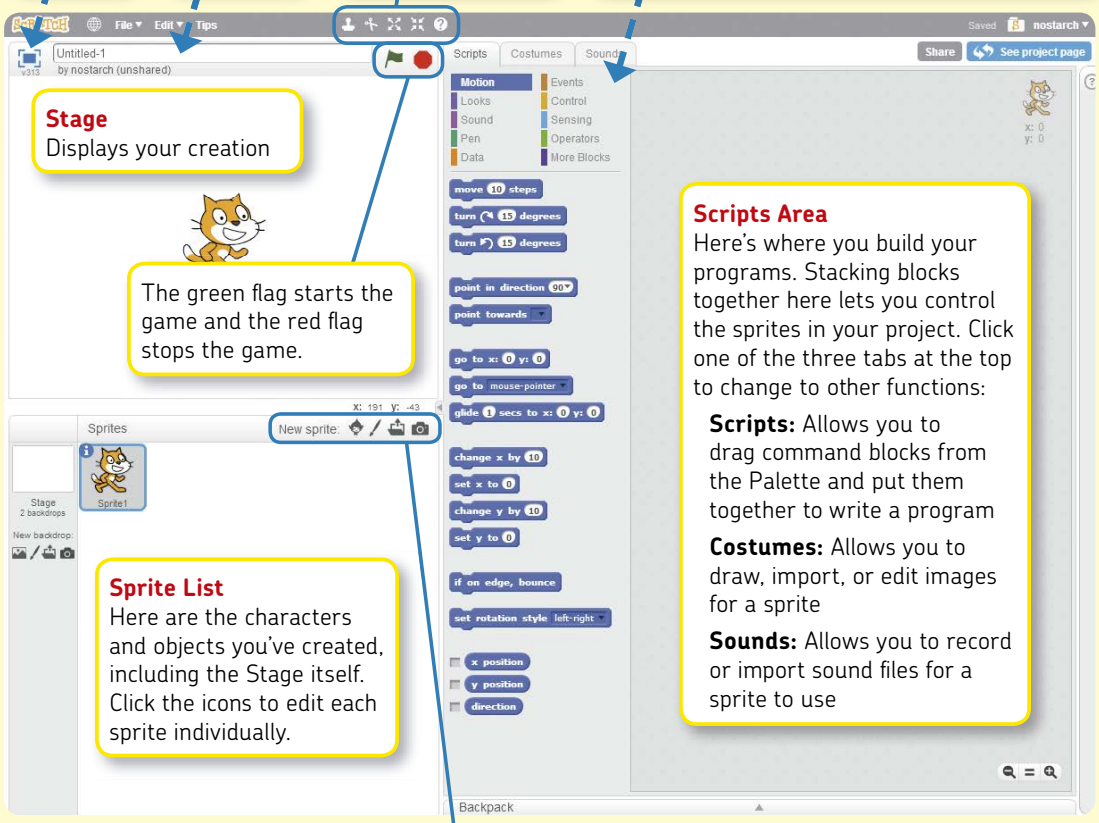


To move a block, just click and drag it over here. This is called the Scripts Area. It's where we write our programs.

You'll need to give each sprite its own instructions. In other games we play, we'll have more than one character to control, so we'll have more than one sprite listed here, in the Sprite List. To give a particular sprite instructions, click a sprite in the Sprite List first and then drag blocks into the Scripts Area.

Now let's take a closer look at the rest of the interface...

A Guided Tour of the Scratch Interface!



CODING IPHONE APPS FOR KIDS

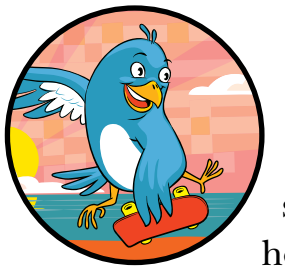
A PLAYFUL INTRODUCTION TO SWIFT

GLORIA WINQUIST AND MATT MCCARTHY

no starch press

3

MAKING CHOICES



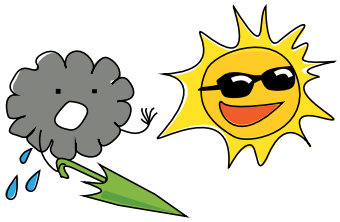
Now that we've covered how to create constants and variables, you're ready to learn how to tell your computer to make choices.

This chapter is about controlling the flow of a computer program by telling the computer which path to take. When we talk about *flow*, we're referring to the order in which the statements of the program are executed.

Up to this point, you've only seen statements performed in the order you've typed them. You've done some cool things with this, but by telling the computer how to make choices about the order of executing

statements, you can do even more. To get the computer to make a choice, we'll use *conditional statements* that tell the computer to run some code based on a condition's value.

You already use conditional statements to make choices every day! For example, before you leave home in the morning, you check the weather. If it's sunny, you may put on a pair of sunglasses. If it's raining, you grab your umbrella. In each case, you're checking a condition. If the condition "it is raining" is true, then you take your umbrella when you leave the house. When the condition could be true or false, it's called a *Boolean expression*. The `Bool` data type that you learned about in Chapter 2 is used to represent the value `true` or `false`.



BOOLEAN EXPRESSIONS

A common type of Boolean expression is one that compares two values using a *comparison operator*. There are six comparison operators. Let's start with two simple ones: *is equal* and *is not equal*.

IS EQUAL AND IS NOT EQUAL

You'll use the *is equal* and *is not equal* comparison operators a lot. *Is equal* is written with two equal signs next to each other, like this: `==`. *Is not equal* is written with an exclamation mark and one equal sign, like this: `!=`.

Let's try them both out in the playground!

```
❶ 3 + 2 == 5                                                    true
  4 + 5 == 8                                                    false
❷ 3 != 5                                                        true
  4 + 5 != 8                                                    true
  // This is wrong and will give you an error
❸ 3 + 5 = 8                                                     error
```

In plain English, the line at ❶ says, "three plus two equals five," which is a `true` statement, and the output in the right pane will confirm this as soon as you finish typing it. At ❷, the line says, "three is not equal to five," which is also a `true` statement. Note that ❸ is an error. Do you know why? While `=` and `==` look a lot alike, remember that a single equal sign (`=`) assigns values. That statement reads, "Put the value of 8 into something called 3 + 5," which doesn't work.

In Swift, the `==` operator also works with other data types, not just numbers. Let's try making some other comparisons.

```
// Comparing strings
let myName = "Gloria"                                   "Gloria"
myName == "Melissa"                                     false
myName == "Gloria"                                      true
❶ myName == "gloria"                                    false
var myHeight = 67.5                                     67.5
myHeight == 67.5                                        true
// This is wrong and will give you an error
❷ myHeight == myName                                    error
```

The line at ❶ is a tricky one; did you expect it to be `true`? Those two strings are close but not exactly the same, and an *is equal* comparison is `true` only if the two values match exactly. The constant `myName` has a value of `"Gloria"` with a capital *G*, which is not the same as `"gloria"` with a lowercase *g*.

Remember in Chapter 2 when we said that you can't use math operators like `+` and `*` on things that aren't the same data type? The same is true for comparisons. You can't compare things that are different types. The line at ❷ will cause an error because one is a `String` and the other is a `Double`.

GREATER THAN AND LESS THAN

Now let's look at four other comparison operators. We'll start with *greater than* (written as `>`) and *less than* (written as `<`). You probably already have a good idea of how these work. A Boolean expression like `9 > 7`, which reads "9 is greater than 7," is `true`. Often, you'll also want to know if something is greater than or equal to something or less than or equal to something. There are two more operators that cover those cases: *greater than or equal to* (which looks like `>=`) and *less than or equal to* (which looks like `<=`). Let's try these out with some more examples:

```
// Greater than
9 > 7                                                   true
// Less than
9 < 11                                                  true
// Greater than or equal to
❶ 3 + 4 >= 7                                            true
❷ 3 + 4 > 7                                             false
// Less than or equal to
❸ 5 + 6 <= 11                                           true
❹ 5 + 6 < 11                                            false
```

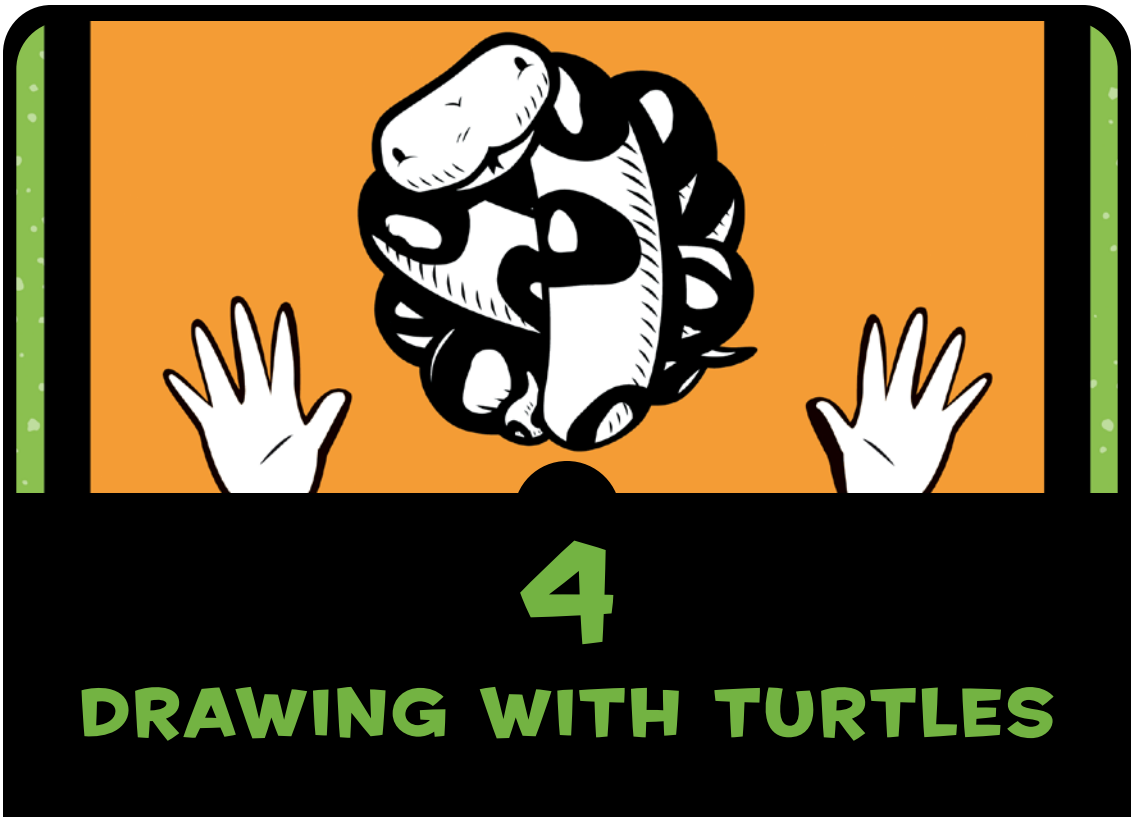
Note the difference between *greater than or equal to* at ❶ and *greater than* at ❷. The sum of `3 + 4` is not greater than `7`, but it is greater than or equal to `7`. Similarly, `5 + 6` is less than or equal to `11` ❸, but it's not less than `11` ❹.

PYTHON FOR KIDS

A PLAYFUL INTRODUCTION TO PROGRAMMING

JASON R. BRIGGS

no starch press



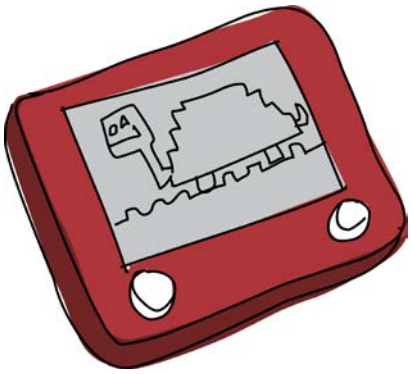
4
DRAWING WITH TURTLES

A *turtle* in Python is sort of like a turtle in the real world. We know a turtle as a reptile that moves around very slowly and carries its house on its back. In the world of Python, a turtle is a small, black arrow that moves slowly around the screen. Actually, considering that a Python turtle leaves a trail as it moves around the screen, it's actually less like a turtle and more like a snail or a slug.

The turtle is a nice way to learn some of the basics of computer graphics, so in this chapter, we'll use a Python turtle to draw some simple shapes and lines.

USING PYTHON'S TURTLE MODULE

A *module* in Python is a way of providing useful code to be used by another program (among other things, the module can contain functions we can use). We'll learn more about modules in Chapter 7. Python has a special module called `turtle` that we can use to learn how computers draw pictures on a screen. The `turtle` module is a way of programming vector graphics, which is basically just drawing with simple lines, dots, and curves.



Let's see how the turtle works. First, start the Python shell by clicking the desktop icon (or if you're using Ubuntu, select **Applications ▸ Programming ▸ IDLE**). Next, tell Python to use the turtle by importing the `turtle` module, as follows:

```
>>> import turtle
```

Importing a module tells Python that you want to use it.

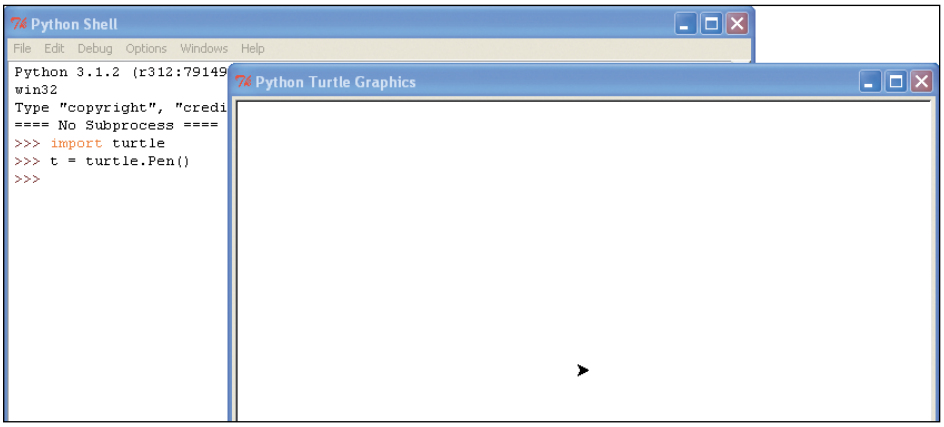
NOTE *If you're using a version of Ubuntu older than 14.04 and you get an error at this point, you might need to install `tkinter`. To do so, open the Ubuntu Software Center and enter* ***`python-tk`*** *in the search box.* Tkinter – Writing Tk Applications with Python *should appear in the window. Click* ***Install*** *to install this package.*

CREATING A CANVAS

Now that we have imported the `turtle` module, we need to create a canvas—a blank space to draw on, like an artist's canvas. To do so, we call the function `Pen` from the `turtle` module, which automatically creates a canvas (we'll learn more about what a function is later). Enter this into the Python shell:

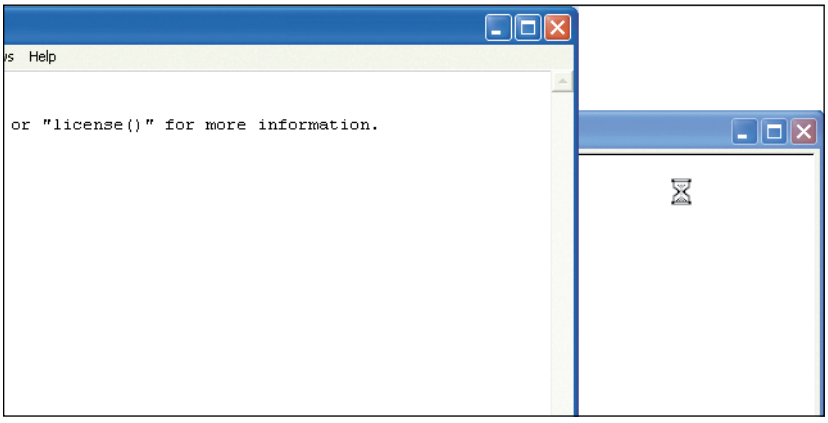
```
>>> t = turtle.Pen()
```

You should see a blank box (the canvas), with an arrow in the center, something like this:



The arrow in the middle of the screen is the turtle, and you're right—it isn't very turtle-like.

If the Turtle window appears behind the Python Shell window, you may find that it doesn't seem to be working properly. When you move your mouse over the Turtle window, the cursor turns into an hourglass, like this:

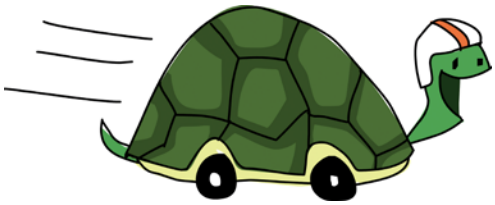


This could happen for several reasons: you haven't started the shell from the icon on your desktop (if you're using Windows or a Mac), you clicked IDLE (Python GUI) in the Windows Start menu,

or IDLE isn't installed correctly. Try exiting and restarting the shell from the desktop icon. If that fails, try using the Python console instead of the shell, as follows:

- In Windows, select **Start ▸ All Programs**, and then in the **Python 3.2** group, click **Python (command line)**.
- In Mac OS X, click the Spotlight icon at the top-right corner of the screen and enter *Terminal* in the input box. Then enter *python* when the terminal opens.
- In Ubuntu, open the terminal from your **Applications** menu and enter *python*.

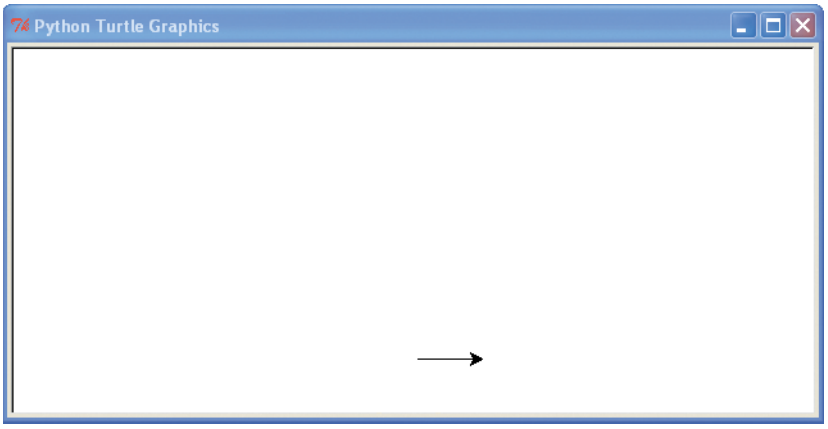
MOVING THE TURTLE



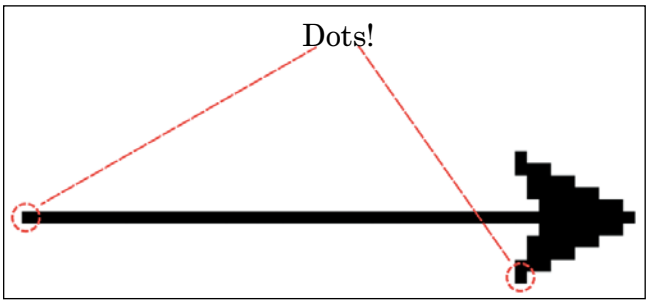
You send instructions to the turtle by using functions available on the variable `t` we just created, similar to using the `Pen` function in the `turtle` module. For example, the `forward` instruction tells the turtle to move forward. To tell the turtle to advance 50 pixels, enter the following command:

```
>>> t.forward(50)
```

You should see something like this:



The turtle has moved forward 50 pixels. A *pixel* is a single point on the screen—the smallest element that can be represented. Everything you see on your computer monitor is made up of pixels, which are tiny, square dots. If you could zoom in on the canvas and the line drawn by the turtle, you would be able to see that the arrow representing the turtle's path is just a bunch of pixels. That's simple computer graphics.



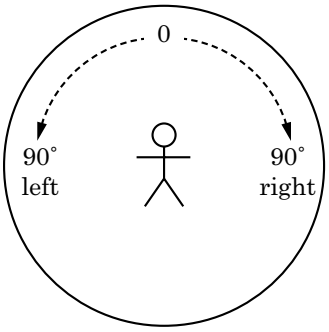
Now we'll tell the turtle to turn left 90 degrees with the following command:

```
>>> t.left(90)
```

If you haven't learned about degrees yet, here's how to think about them. Imagine that you're standing in the center of a circle.

- The direction you're facing is 0 degrees.
- If you hold out your left arm, that's 90 degrees left.
- If you hold out your right arm, that's 90 degrees right.

You can see this 90-degree turn to the left or right here:



LEARN TO PROGRAM WITH MINECRAFT®

TRANSFORM YOUR WORLD WITH THE POWER OF PYTHON

CRAIG RICHARDSON

no starch press

3

BUILDING QUICKLY AND TRAVELING FAR WITH MATH

In Chapter 2, you learned how to create a variable and change its value. In this chapter, you'll learn how to use math in Python to generate any block you want and quickly build complex structures in your Minecraft world. You'll even give yourself superpowers to make the player super jump!

EXPRESSIONS AND STATEMENTS

When you're having a conversation with someone, you want them to understand what you're telling them. You use short phrases, such as "three diamonds" or "behind a tree," to give information to the person you're talking to. However, the phrases don't make sense on their own unless they're combined into sentences, such as "I found three diamonds behind a tree."

Python programming has concepts similar to phrases and sentences, which are called expressions and statements.

You can combine values, variables, and operators to create small pieces of code called *expressions*, like `2 + 2`. Expressions can be combined into *statements*, which you learned about in Chapter 2. Statements are single lines or short blocks of code that do something in a program, such as `zombies = 2 + 2`. In this example, `2 + 2` is an expression and is part of the statement `zombies = 2 + 2`.

For longer programs that use a text editor instead of the Python shell, be sure to write entire statements. For example, the Python shell and a program written in a text editor will treat the expression `2 + 2` entirely differently. When you're using the Python shell in IDLE, Python will output `4` as the result of `2 + 2`, as shown here:

```
>>> 2 + 2
4
```

However, when you're using a text editor, Python won't do anything with the expression because it's not part of a complete statement. To turn this expression into a complete statement, you could assign its value to a variable, like this:

```
zombies = 2 + 2
```

Then print that variable to see its value:

```
print(zombies)
```

When you run this code, it will print `4`.

Again, when writing programs in the text editor, it's very important that you use full statements, not just expressions.

OPERATORS

In math, *operators* are used to alter and combine numbers. For example, the addition operator lets you add two (or more) numbers, and the subtraction operator is used to subtract one number from another.

Python uses all the basic math operators that you already know—addition, subtraction, multiplication, and division—as well as more advanced operators, like exponents. Let's start with addition.

ADDITION

In Python, addition looks like you would normally write it using the plus sign (`+`). For example, if you have two flowers and you pick two more, you could describe that with a statement using addition:

```
>>> flowers = 2 + 2
```

Python works out the result of the expression on the right side of the equal sign and then assigns it to the variable on the left. In this case, the result of the expression on the right is `4`. For the rest of the time that this particular code is in use, the variable `flowers` will have a value of `4`.

You can use addition in Minecraft to build things in the blink of an eye. Are you ready for your next mission? Let's get started!

MISSION #5: STACK BLOCKS

You can use the `setBlock()` function to create and place a block in Minecraft. Just like `setPos()` and `setTilePos()`, `setBlock()` takes x-, y-, and z-coordinates as arguments, but it also needs a fourth value: the block type. This value identifies the kind of block you want to place in the game.

Whether it's grass, lava, melon, or any other block, each type is represented by a specific integer. For example, grass is `2`, empty air is `0`, water is `8`, and melon is `103`. For a full list of blocks and their integer values, see "Block ID Cheat Sheet" on page 299.

To use `setBlock()`, pass values for the x-, y-, and z-coordinates and the integer representing the block type to the function, separated by commas. For example, let's place a melon block (type 103) at coordinates (6, 5, 28):

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.setBlock(6, 5, 28, 103)
```

After the first two familiar lines that you'll see in all Minecraft Python programs, just call `setBlock()` with all the values you want to use. You can also use variables instead of numbers to get the same effect, as shown in Listing 3-1.

blockStack.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
x = 6
y = 5
z = 28
blockType = 103
mc.setBlock(x, y, z, blockType)
```

Listing 3-1: A program to create a melon block

First, create variables to represent the block coordinates (`x`, `y`, and `z`) and type (`blockType`). Then, pass all the variables to the `setBlock()` function, and the Minecraft Python API works its magic. Now you can use those variables again anywhere in your program, and if you decide to change their values later, you only have to change them in one place.

When you combine this code with math operators, you can do some pretty cool things. Let's create a stack of blocks.

Create a new folder called *math* within the *Minecraft Python* folder. Open IDLE and create a blank program using IDLE's text editor. Save this file as *blockStack.py* in the *math* folder. Copy the code from Listing 3-1 into your editor and add the two lines from Listing 3-2 to stack another melon block on top of the one you just set.

blockStack.py

```
❶ y = y + 1
❷ mc.setBlock(x, y, z, blockType)
```

Listing 3-2: Extra code to stack a second melon block on top of the first melon

You're adding 1 to the value of `y` ❶, and you're using the `setBlock()` function to create another new block ❷. By increasing the value of `y` by 1, the second block is placed higher on the y-axis than the first block, so the second block is stacked on top of the first one.

From here, your mission is to add two more blocks to the stack. Try modifying your *blockStack.py* program so it stacks four blocks instead of two! When you run your program, a stack of four melon blocks should appear, as shown in Figure 3-1.

Figure 3-1: I've made a stack of melon blocks.

HINT *To add a second block on top of the first, we increased the* `y` *variable by 1 and then used the* `setBlock()` *function again. What do you think would happen if you reused these two statements at the end of your program? What if you used them three times? Would this be a solution for creating a stack of four blocks?*

BONUS OBJECTIVE: CREATE A RAINBOW

You could write many variations of the *blockStack.py* program. By modifying the block types, you can create a rainbow or a tower of lava! Try changing the block types to see what you can create.

MISSION #6: SUPER JUMP

In Chapter 2, you learned how to change the player's location. Let's take that skill one step further and send the player high into the air using the power of addition. First, find out where the player is by calling `getTilePos()`, as shown in Listing 3-3.

superJump.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

position = mc.player.getTilePos()
x = position.x
y = position.y
z = position.z
```

Listing 3-3: Code to find the player's position

The dot between the `position` variable and the `x`, `y`, and `z` is called *dot notation*. Dot notation is used by certain variables and functions, such as all of the functions you use in the Minecraft Python API (for example, `mc.setTilePos()`). You'll learn more about dot notation in Chapters 11 and 12.

Once you have the player's position, you can set the `x`, `y`, and `z` variables to the player's current coordinates, which are represented by `position.x`, `position.y`, and `position.z`. You can then teleport the player anywhere you want in relation to the current coordinates, as shown in Listing 3-4.

superJump.py

```
x = x + 5
mc.player.setTilePos(x, y, z)
```

Listing 3-4: Code to move the player's x position up by 5 blocks

Here, I've transported the player 5 blocks along the x-axis, but this isn't that special: you can move the player around horizontally any time you want in Minecraft. Let's give the player a super jump instead!

Your mission is to make the player jump 10 blocks into the air above their current position. You should be able to do this using the code in Listings 3-3 and 3-4 but with some slight differences. Copy the code in Listings 3-3 and 3-4 into IDLE, save it as *superJump.py*, and change the `y` variable in a similar way to how I changed the `x` variable. When you run the program, the player should jump into the air, as in Figure 3-2.

THE LEGO® ARCHITECT

TOM ALPHIN

no starch press

TALIESIN WEST
Scottsdale, Arizona, 1937, Frank Lloyd Wright.
LEGO model by Adam Reed Tucker.

MODERNIST LEGO MODELS

VILLA AMANZI
Phuket, Thailand, 2008, Original Vision Ltd.
LEGO model by Robert Turner.

VILLA HILLCREST
LEGO model by Ken, Amethel,
and Kai Parel-Sewell.

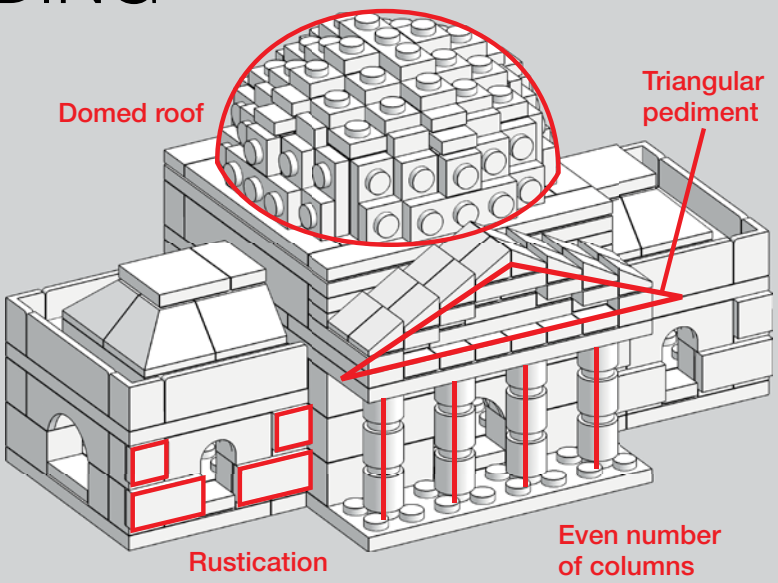
MODERN HOME
LEGO model by Dave Kaleta.

MONTICELLO
Charlottesville, Virginia, 1772,
Thomas Jefferson.

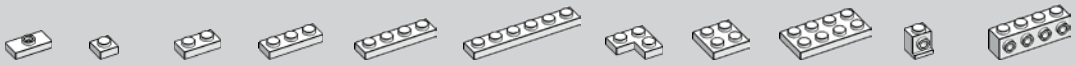
DOMED BUILDING

This model includes many iconic elements of Neoclassical architecture, including a prominent domed roof.

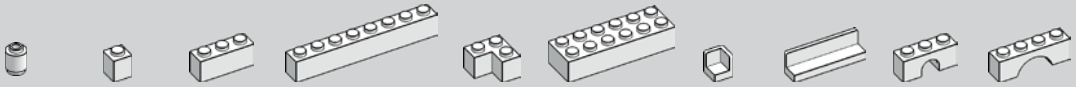
The overall shape is similar to Thomas Jefferson's design for Monticello. This design could be described as Palladian, as it is strictly symmetrical and includes both columns and a pediment.



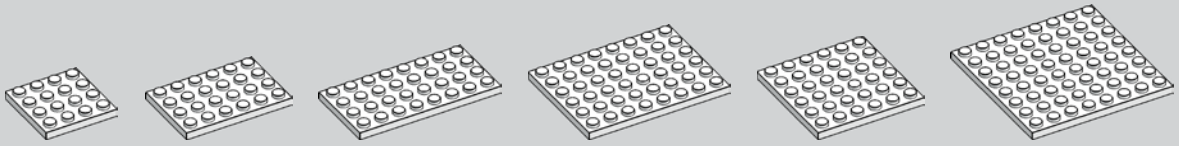
8x	14x	3x	2x	2x	16x	4x	4x
3070	3069	6636	4162	3068	54200	3039	3045



40x	16x	18x	11x	12x	4x	16x	6x	1x	12x	4x
3794	3024	3023	3623	3710	3666	2420	3022	3020	4070	30414



15x	8x	6x	3x	6x	2x	4x	6x	4x	4x
3062a	3005	3622	3008	2357	2456	6231	30413	4490	3659



4x	5x	2x	3x	2x	1x
3031	3032	3035	3036	3958	41539

THE LEGO® TRAINS BOOK

HOLGER MATTHES

no starch press

CASE STUDIES IN DESIGN

Armed with the tools and knowledge about LEGO modeling covered in the previous chapters, we'll now take a closer look at the actual design process using some of my own builds as a guide.

BUILDING A STEAM ENGINE

The biggest challenge in fitting a LEGO train on a track is building a working model of a big steam engine or a historical engine like the Crocodile. The large driving wheels (also known as *main drivers*) are attached to a chassis and are linked with connecting rods and piston rods. When three or more axles with large wheels are fixed to a chassis, the train can't go around curves. You also need to consider leading and trailing axles. They improve drivability on real-life trains, but they can be a problem on LEGO models.

Here are some tips for building these complex locomotives in LEGO:

- Use wheels without a flange ("blind drivers").
- Use wheels that can slide sideways.
- Divide the driving wheels into smaller, pivoted units.
- Use connecting rods that are not connected to all wheels.
- Strategically place pivot points for leading and trailing axles.

The first step is to build a rough model of the chassis to get a feel for the layout of the axles. Then you can work out how and where to place the drive-train. But your main priority should be making sure the train can run through curves and switches.

Complex chassis for a steam engine with 2-8-2 wheel arrangement (BR 41 during development)

ATTACHING LEADING AND TRAILING WHEELS

Once you've sorted out your driving wheels, the next step is attaching the leading and trailing axles to the chassis. These axles need a lot of space to turn left and right to get through turns and switches.

The leading wheels of a steam engine are typically located close to the steam cylinders. If you use moving piston rods, the steam cylinders must be fixed and cannot pivot. And because the leading axle needs to turn, it's critical to keep a lot of space between the steam cylinders. Test your chassis on a curved track to make sure everything runs smoothly.

The challenge for the trailing wheel is making sure it can withstand the push power from the motors in the tender. The trailing wheel must have a stable but flexible attachment.

A fixed leading wheel can be used on a shorter chassis where only one axle of the drive wheels uses regular wheels and the other axles use blind drivers.

Steam cylinders interfere with the leading wheels.

Chassis with fixed leading wheels and blind drivers

USING SPOKED WHEELS FOR STEAM TRAINS

Necessity is the mother of invention—or at least that was the motto in the early 2000s when some train builders needed larger wheels for their steam trains. Starting with the large spoked wheels from some 1970s vintage car sets, some builders simply lathed off the outer flange, and others with better tools created aluminum treads and flanges to fit around the wheels. Those custom flanges and wheels have never been sold officially, but you can find some through LEGO forums.

Vintage LEGO spoked wheel with homemade flange

Since their creation, trains have been a small theme in the LEGO universe, with some fans seeing the theme only as an extension of LEGO Town or City. Nevertheless, the LEGO Group has continually improved its trains over the years.

Just as old and new bricks fit together, older and newer LEGO trains are compatible. This is because the *gauge*, or width, of the track has remained the same year after year. What's changed is how LEGO trains are powered.

The earliest LEGO trains were released in 1966, beginning the *Blue Era*. This era included unpowered trains, 4.5 V battery-powered trains, and a 12 V power system. This system maintained the same plastic running rails but introduced a clever inner metal rail placed between the plastic rails. This adaptation allows unpowered, 4.5 V powered, and 12 V powered trains to seamlessly share the same track. In 1980, the *Gray Era* introduced improvements to the power components, as well as new gray track.

Locomotive with motor from an early set (#112)

A decade later, LEGO completely changed the architecture of the train system by introducing all-metal track components powered by a new 9 V electrical system. By 2006, this 9 V metal track system was replaced by the more economical plastic RC track system. The plastic RC track was nearly identical to (and compatible with) the metal 9 V track it replaced. Today's Power Functions trains have reverted to battery-powered operation but can be controlled remotely by infrared.

This chapter gives an overview of the various eras of LEGO trains, their key components, and how they fit together.

Four decades lie between the oldest (#182, right) and newest (#10233, left) LEGO train sets in this lineup.

THE LEGO® MINDSTORMS® EV3 DISCOVERY BOOK

a beginner's guide to building and programming robots

laurens **valk**

no starch press

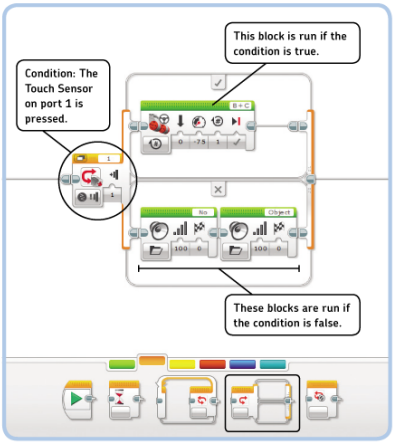


Figure 6-12: The Switch block checks whether the condition is true or false and runs the appropriate blocks. You specify the condition using the mode and settings on the Switch block.

configuring a switch block

You define the condition by configuring the mode and settings of the Switch block. Once the program arrives at the Switch block, the robot checks whether the condition is true. Then, it decides which set of programming blocks in the switch to run.

There's a mode for each sensor; in this case, you'll choose the one for the Touch Sensor, namely **Touch Sensor – Compare – State** (the only available option). Once you've chosen this mode, you can specify in the state setting whether the Touch Sensor must be pressed (1) or released (0) for the condition to be true. As before, set Port to **1** to specify how the Touch Sensor is connected to your EV3.

sensors and the switch block in action

The *TouchSwitch* program you'll now create makes the robot drive forward for three seconds. Then, if the Touch Sensor is pressed, the robot reverses for a short while. If the sensor is not pressed, the robot instead says "No Object." Finally, regardless of the Switch block's decision, the robot plays a tone. Now create the program, as shown in Figure 6-13.

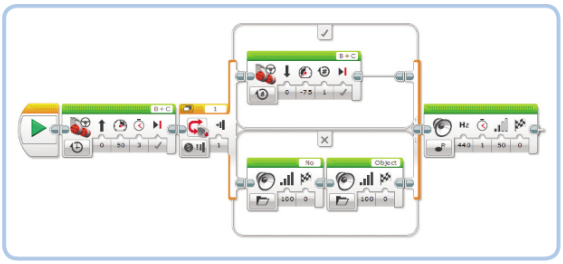


Figure 6-13: The TouchSwitch program has the robot decide what to do based on a sensor reading.

Try running this program a few times, and determine when you need to press the Touch Sensor to make the robot go backward. Your experiments should show that the robot takes a measurement when the Switch block runs and that it uses this single measurement to determine whether the condition is true. In this program, the sensor measurement is taken just after the robot finishes going forward. When either the reverse action or the "no object" action is complete, the tone plays.

adding blocks to a switch block

There's no limit to the number of blocks you can place inside a Switch block. If one part of a switch has multiple blocks, they're simply run one by one. You can also leave one of the two parts of a Switch block empty, as shown in Figure 6-14.

Run this modified program to see what happens. If the condition is true (the bumper is pressed), the robot should say "Object" and move backward, and the program should continue by playing the tone. If the condition is false (the sensor is not pressed), the program will find no blocks in the lower part of the switch and instantly move on to the Sound block after the switch.

DISCOVERY #27: STAY OR MOVE?

Difficulty: **Time:**

Make the robot stand still for three seconds. Then, if the Touch Sensor is released, the robot should turn around and drive forward for five wheel rotations. But if the sensor is pressed, the robot should do nothing, and the program should end immediately.

DISCOVERY #28: DIFFICULT DECISIONS!

Difficulty: **Time:**

Let's practice with the Switch block! Create a program to implement the decision tree shown in Figure 6-15. How do you configure the Switch block, and why do you have to put a Wait block at the end of the program?

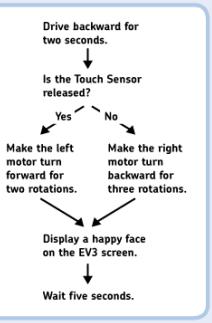


Figure 6-15: The program flow for Discovery #28

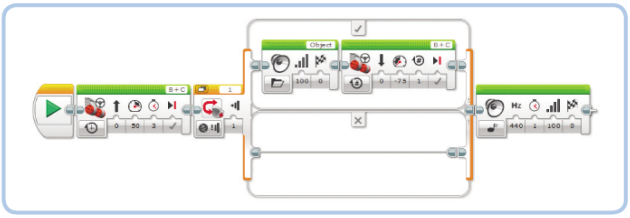
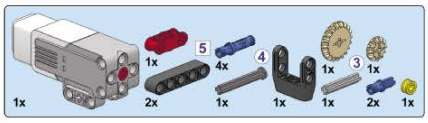
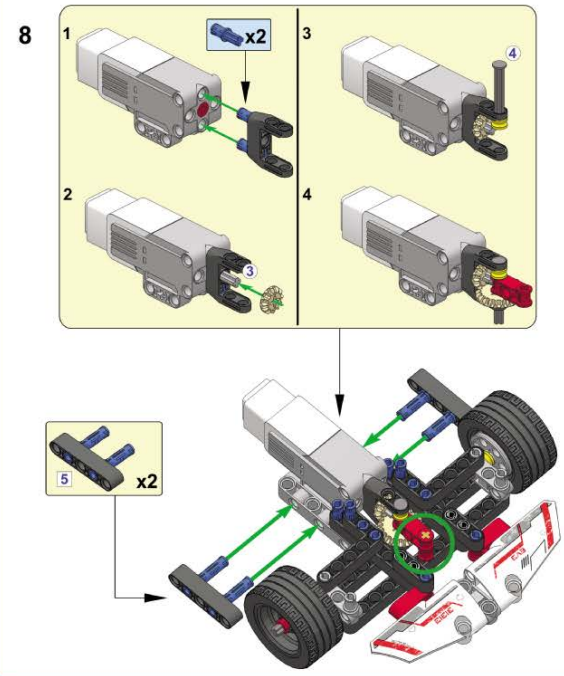
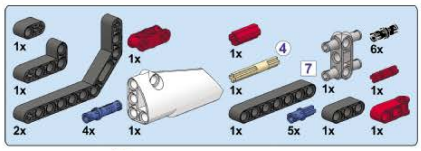
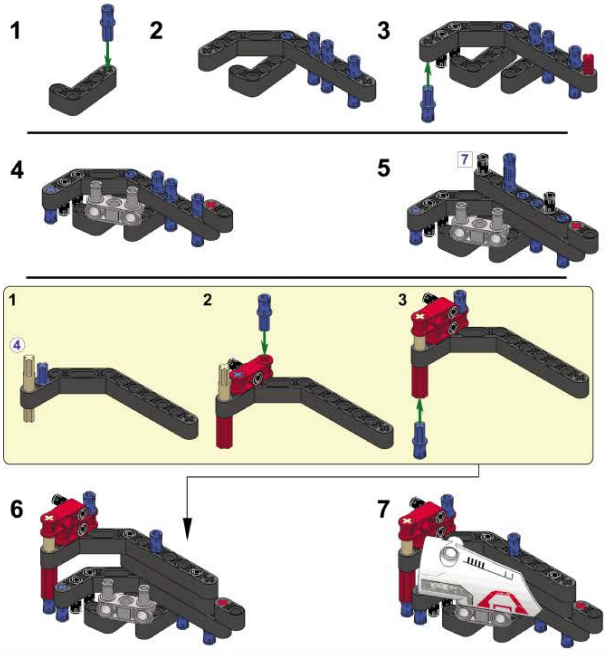


Figure 6-14: A modified version of the TouchSwitch program. The switch does not have any blocks to run if the condition is false, so the program immediately plays a tone after moving forward if the sensor is not pressed.









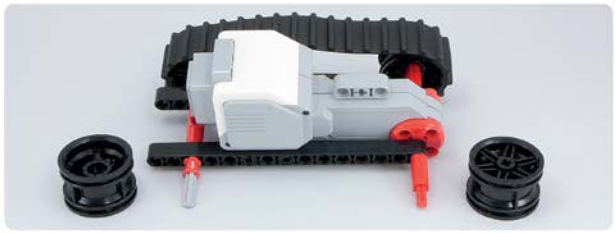
THE LEGO® MINDSTORMS® EV3 IDEA BOOK

YOSHIHITO ISOGAWA

181 Simple Machines and Clever Contraptions

Crawlers

#115



#116

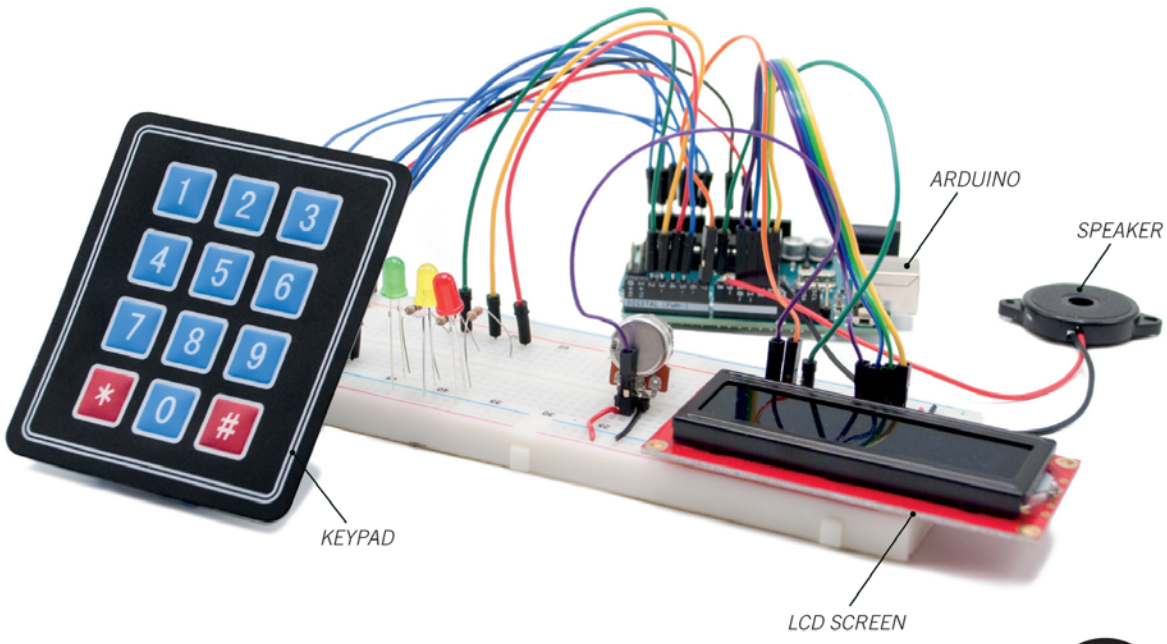
Walking machines

#122

ARDUINO PROJECT HANDBOOK

25 SIMPLE ELECTRONICS PROJECTS FOR BEGINNERS

VOLUME 2

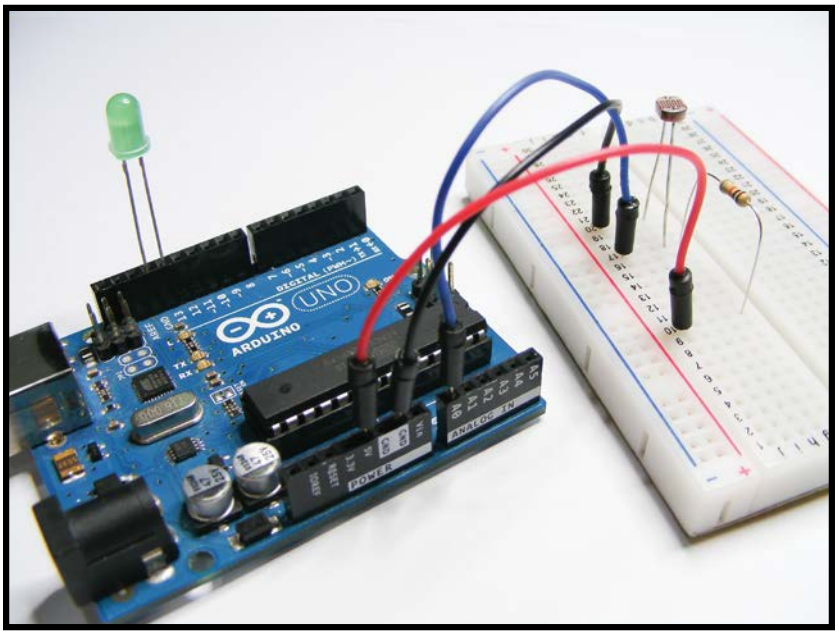


MARK GEDDES

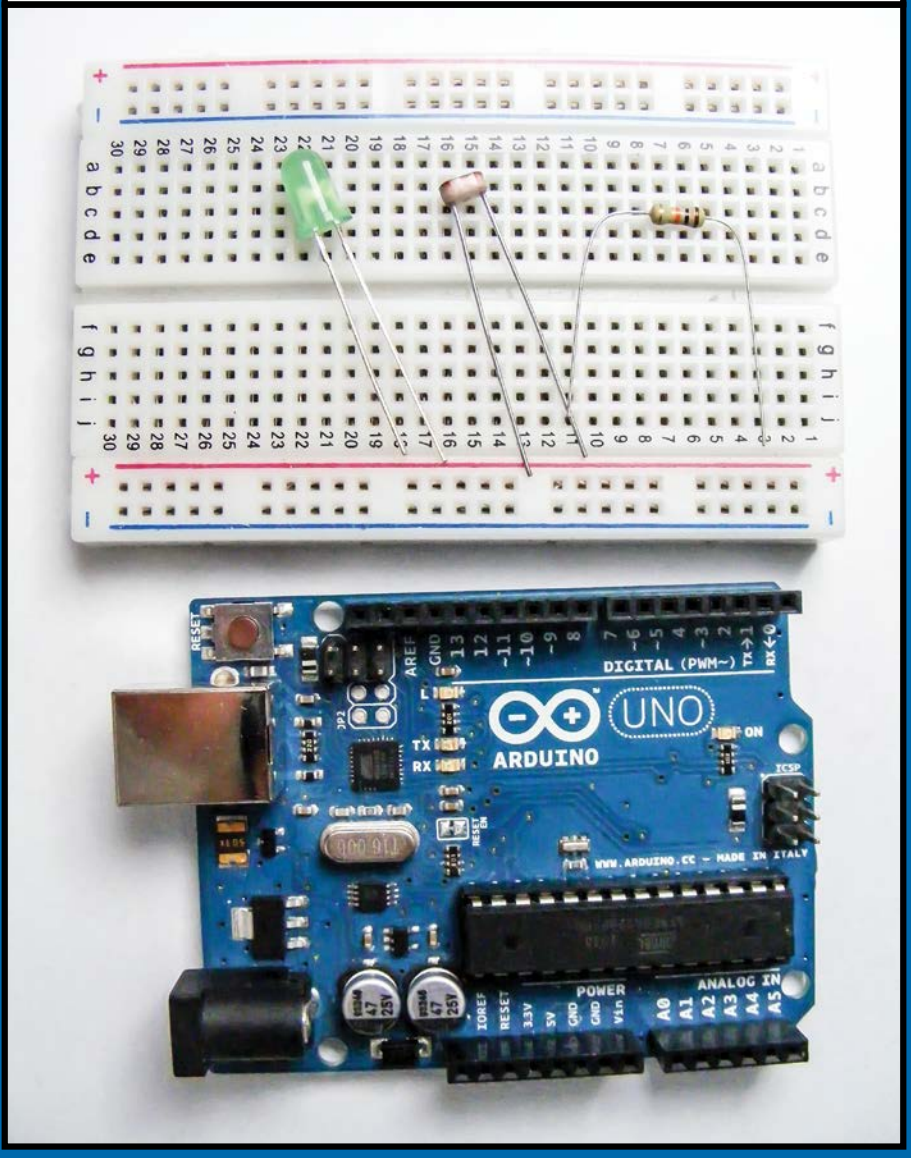
no starch press

2
LIGHT-ACTIVATED NIGHT-LIGHT

THIS PROJECT IS A SIMPLE TEST OF A PHOTORESISTOR'S FUNCTIONALITY: WE'LL CREATE A NIGHT LIGHT THAT GETS BRIGHTER DEPENDING ON THE AMOUNT OF LIGHT DETECTED.



COST: $ TIME: 10 MINUTES



PARTS REQUIRED

Arduino board
Breadboard
Jumper wires
Photoresistor
LED
10k-ohm resistor

HOW IT WORKS

A *photoresistor* is a variable resistor that reacts to light; the less light that shines on it, the higher the resistance it provides. This resistance value varies the voltage that's sent to the input pin of the Arduino, which in turn sends that voltage value to the output pin as the power level of the LED, so in low light the LED will be bright. There are different styles of photoresistors, but they usually have a small, clear, oval head with wavy lines (see Figure 2-1). Photoresistors do not have polarity, so it doesn't matter which way you connect the legs.

The principles at work here are similar to those of a child's nightlight. You can use a photoresistor to control more than just LEDs, as we'll see in upcoming chapters. Since we only have two power and GND connections, we won't be using the breadboard power rails here.

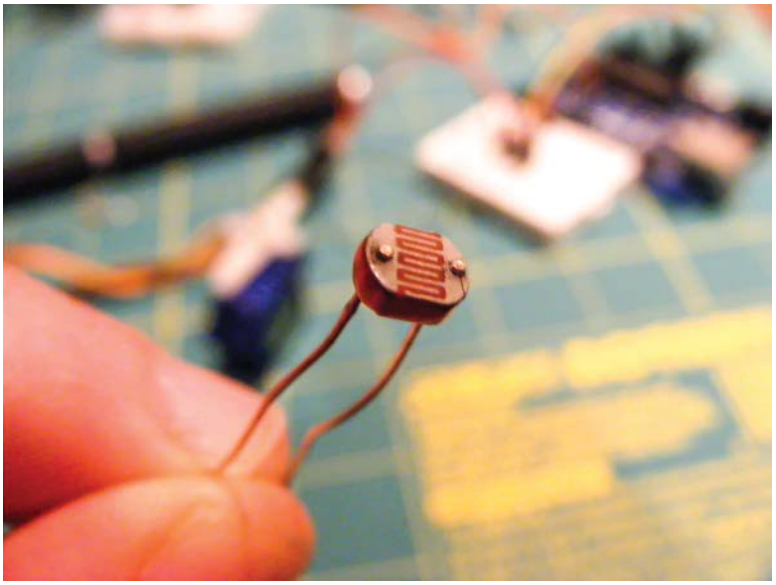


FIGURE 2-1:
A photoresistor

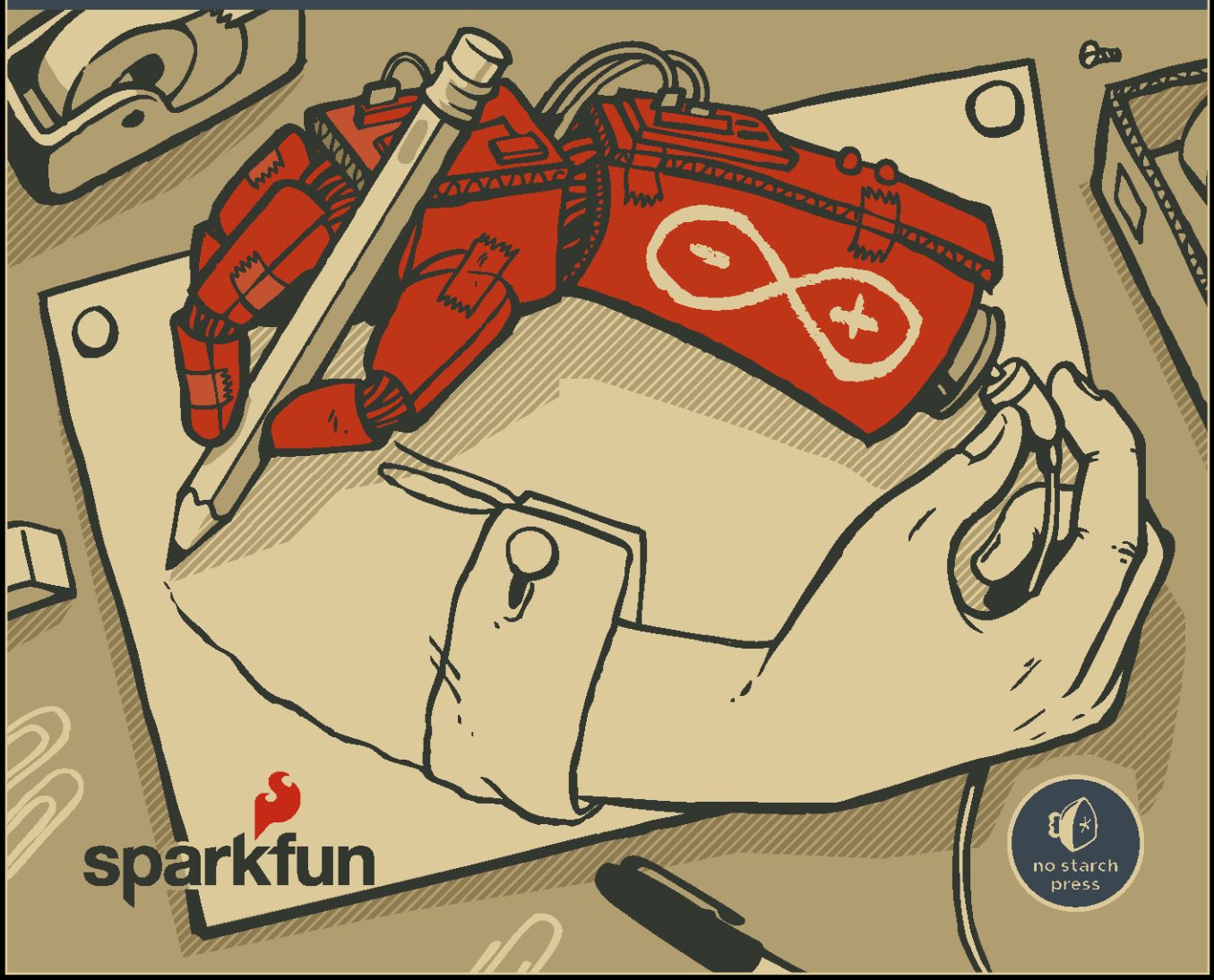
THE BUILD

1. Place your photoresistor in the breadboard, connecting one leg to GND directly on the Arduino and the other leg to Arduino A0.
2. Connect one leg of the 10k-ohm resistor to +5V, and connect the other leg to the A0 photoresistor leg, as shown in the circuit diagram in Figure 2-2.

THE ARDUINO INVENTOR'S GUIDE

LEARN ELECTRONICS BY MAKING 10 AWESOME PROJECTS

BRIAN HUANG AND DEREK RUNBERG



sparkfun

no starch press

TINY DESKTOP GREENHOUSE

GREENHOUSES COME IN ALL SHAPES AND SIZES, FROM TINY INDOOR GREENHOUSES MADE FROM PLASTIC SHEETING TO LARGE INDUSTRIAL GREENHOUSES THAT SPAN THOUSANDS OF SQUARE FEET. NOT EVERYONE WANTS A FULL-SIZE GREENHOUSE, THOUGH, SO IN THIS PROJECT YOU'LL BUILD A SMALLER-SCALE MODEL THAT CAN SIT ON YOUR DESK (FIGURE 7-1).

BUILD THE SERVO MOTOR AUTOVENT

You'll use a servo motor like the one you used in Project 6 to open and close a window on the greenhouse. A servo is a simple motor that uses three wires for the control signal (yellow or white), power (red), and ground (black) connections, as shown in Figure 7-12.

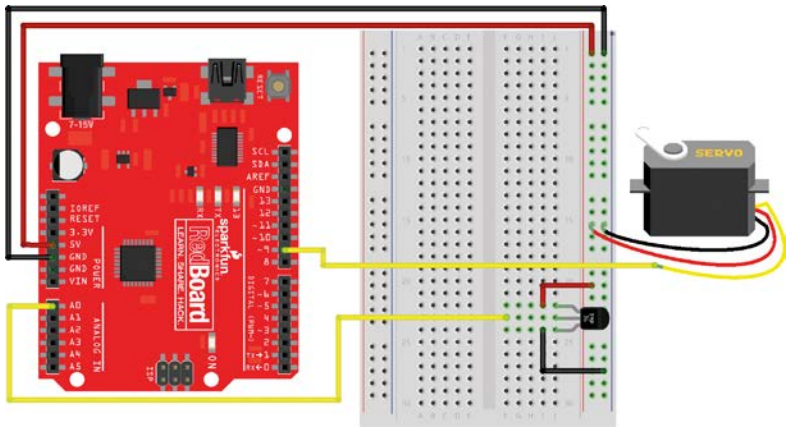


FIGURE 7-12: Adding the servo motor to the circuit

Most standard servos have a three-pin female connector. Connect three male-to-male jumper wires to these three pins to extend these connections, as shown in Figure 7-13. If possible, match the colors of your jumper wires to the leads on the servo connector. Then, connect the yellow (or white) signal wire to pin 9 on the Arduino board. Connect the red wire to 5 V and the black wire to GND on the right side of the breadboard.

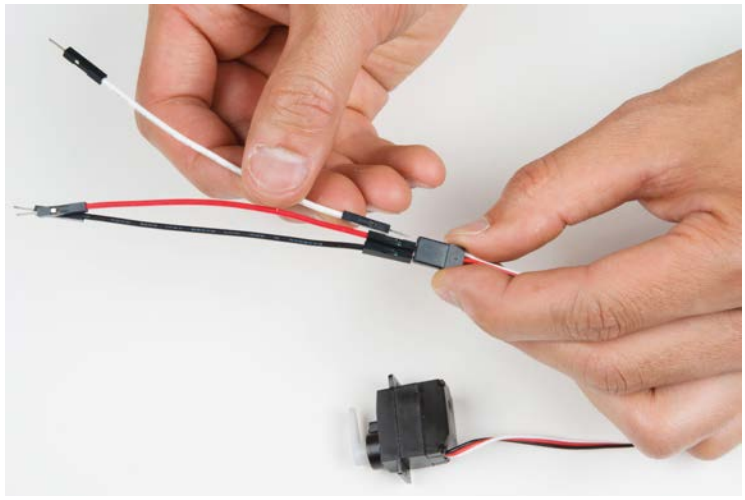


FIGURE 7-13: Inserting male-to-male jumper wires to connect the servo to the breadboard

PROGRAM THE AUTOVENT

Add the servo code shown in Listing 7-5.

```
❶ #include<Servo.h>
❷ Servo myServo;

  //Example sketch - calculates the temperature from the TMP36
  //sensor and prints it to the Serial Monitor
  int rawSensorValue;
  float rawVolts;
  float tempC;
  float tempF;
❸ int setPoint = 85;
  int returnPoint = 83;

  void setup()
  {
    myServo.attach(❹9, 1000, 2000); //initializes myServo object
    Serial.begin(9600);  //initializes the serial communication
    --snip--
  }

  void loop()
  {
    --snip--
    Serial.println();    //new line character

❺   if(tempF > setPoint)
    {
      myServo.write(180);
    }
    else if(tempF < returnPoint)
    {
      myServo.write(0);
    }
    delay(1000);
  }

  /*****************************************************/
  float volts(int rawCount)
  --snip--
```

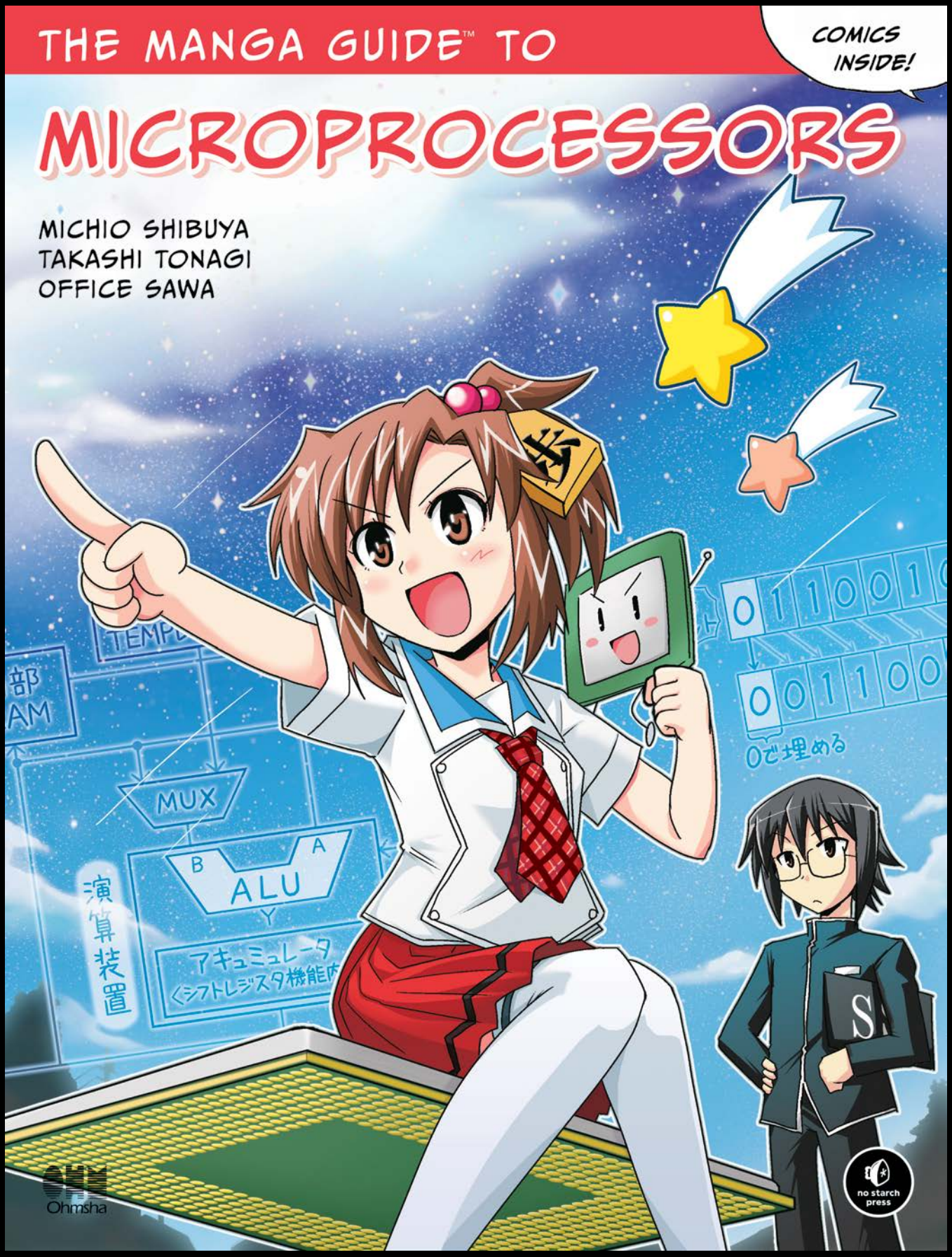
LISTING 7-5:
Adding in the servo control

Remember from Project 6 that to use the servo you need to include the Servo library ❶ and create a servo object named `myServo` ❷. Next, create two variables ❸ to define the *set points* for the control system. These set points are the temperatures

THE MANGA GUIDE™ TO MICROPROCESSORS

COMICS INSIDE!

MICHIO SHIBUYA
TAKASHI TONAGI
OFFICE SAWA



Ohmsha
no starch press

THE RECIPROCAL STATES OF 1 AND 0

OKAY, LET ME START OFF WITH A QUESTION!

LAST TIME, YOU SAID, "COMPUTERS LIVE IN A WORLD OF 1s AND 0s," BUT THAT WAS ALL PRETTY ABSTRACT.

AYUMI

WHAT DO YOU MEAN BY 1s AND 0s ANYWAY?

GOOD QUESTION... YOU CAN THINK OF 1s AND 0s AS TWO RECIPROCAL STATES THAT ARE OPPOSITES.

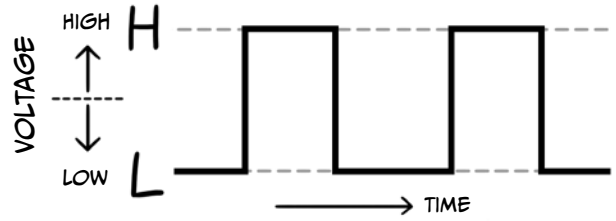
0/1

TWO RECIPROCAL STATES...

THEY'RE MORE LIKE INDICATORS THAN NUMBERS, REALLY.

YOU MEAN LIKE LIGHT AND DARK, LIFE AND DEATH, OR ON AND OFF?

PRECISELY!

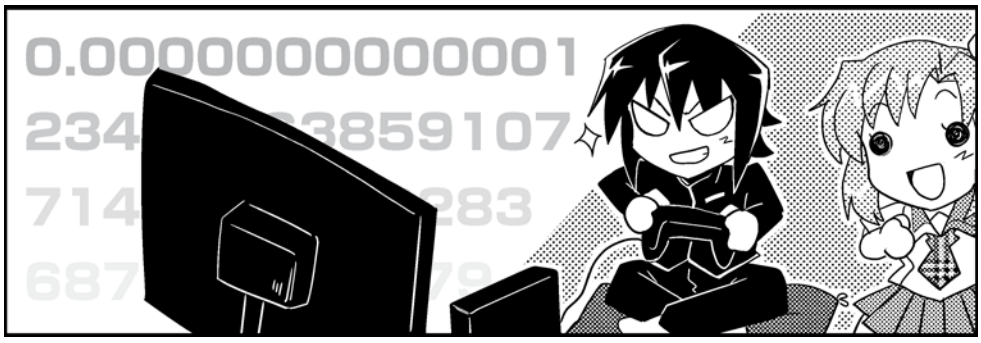


VOLTAGE CHANGES WITH TIME

TO PUT IT ANOTHER WAY, THE VOLTAGES IN COMPUTER CIRCUITS GENERALLY FALL INTO TWO BANDS. HIGH VOLTAGES ARE CLOSE TO THE SUPPLY VOLTAGE, AND LOW VOLTAGES ARE CLOSE TO GROUND.*

* *GROUND* IS THE REFERENCE POINT FOR VOLTAGE AND IS EQUIVALENT TO ZERO VOLTS.

FIXED-POINT AND FLOATING-POINT FRACTIONS



Next up, I'll teach you a really important concept. In computers, there are two ways to store fractions—either fixed point or floating point.

When using extremely small values like 0.00000000000000 . . . 001 or very large values like 1000000000000000 . . . , it's a lot more practical to use floating-point fractions.

Hmm . . . why is that? What's the difference?

Well, for example, instead of writing a billion in decimal as 1,000,000,000, you could write it as 10^9 to save some space, right? And if you had a number like 1,230,000,000, you could represent it as 1.23×10^9 instead. We call this form *scientific notation* or *standard form*, where the n in 10^n is called the *exponent* and the 1.23 is called the *significand*. *Floating-point* numbers use scientific notation when storing values.

In contrast, *fixed-point* numbers express values the way we're used to, with a decimal point. When expressing integers with this method, you can imagine the decimal point being at the far right of the number. Here's a comparison of the two.

FIXED POINT	FLOATING POINT
123. ← DECIMAL POINT	1.23×10^2
1230000.	1.23×10^6
0.00000123	1.23×10^{-6}

Oh, okay. So if you're using fixed-point fractions to express really large or really small numbers, the number of digits you need increases by a lot. But if you're using floating-point, only the exponent gets bigger or smaller while the number of digits stays the same. Yeah, that's really useful!

That's right. That last example was in decimal, but since computers use binary, the principle becomes even more relevant. The most common variant used is this one.

An example of floating-point representation inside a computer (using a base 10 number as the significand for illustration)

I used the decimal 1.69 just to make it easier to understand. The number would be in binary in a computer. The important part here is that this significand always has to be greater than 1 and less than 2.

Hm . . . so this representation makes it easy for computers to handle extremely small and extremely large numbers. They're also easy to use in calculations, right?

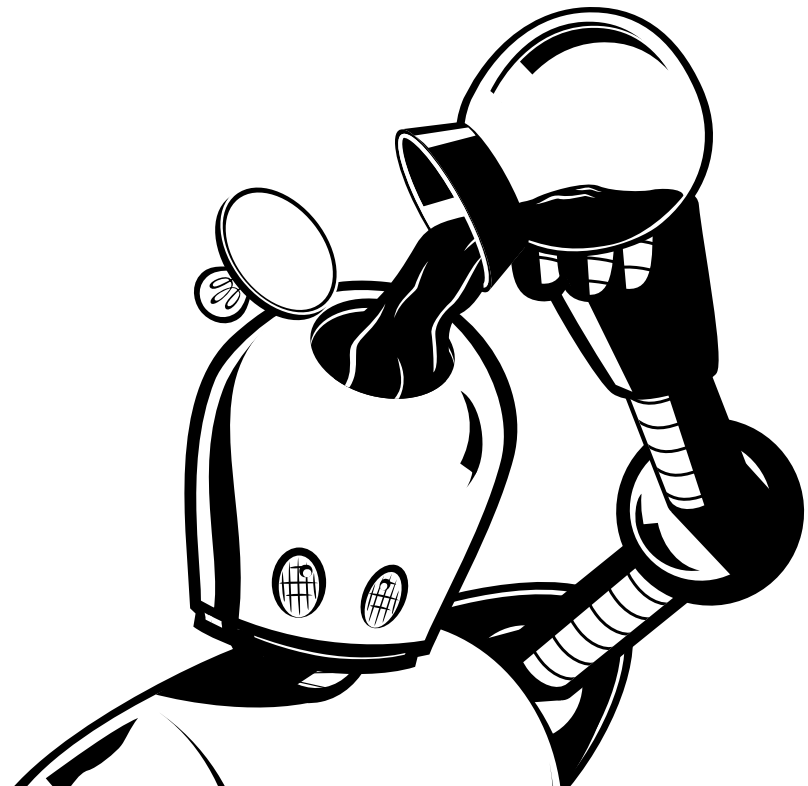
Yes! And it's also important to understand that the speed with which you can calculate using floating-point numbers is critical to CPU performance. Gaming systems that process real-time, high-fidelity graphics also use floating-point arithmetic extensively. (See "CPU Performance Is Measured in FLOPS" on page 137 for a more detailed explanation.)

Generally, scientific calculations require an accuracy of only around 15 digits, but in some cases, 30 are used. Some modern encoding algorithms even use integers of up to 300 digits!

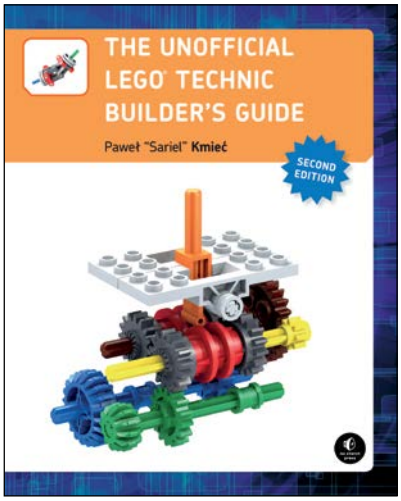
Ugh . . . I don't think I could do those calculations in my head. I hate to lose to computers, but I hope they're at least advancing some fields of science!

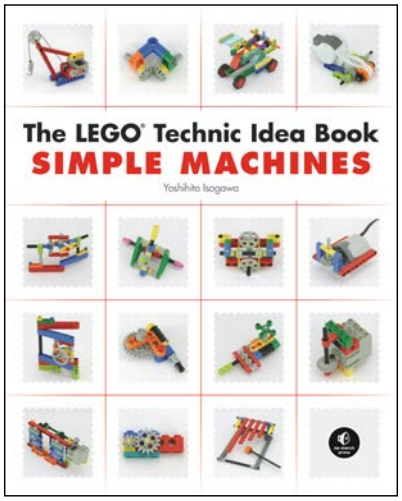
Founded in 1994, No Starch Press is one of the few remaining independent technical book publishers. We publish the finest in geek entertainment—unique books on technology, with a focus on open source, security, hacking, programming, alternative operating systems, and LEGO. Our titles have personality, our authors are passionate, and our books tackle topics that people care about.

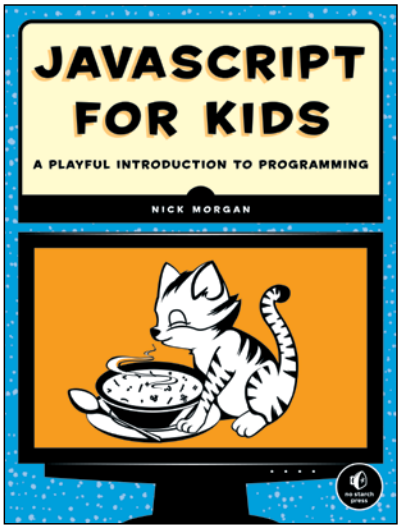
VISIT WWW.NOSTARCH.COM FOR A COMPLETE CATALOG.

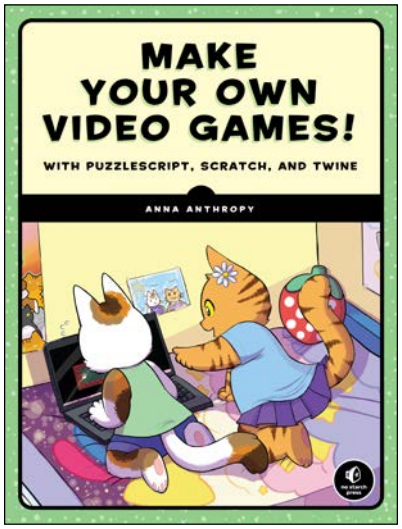


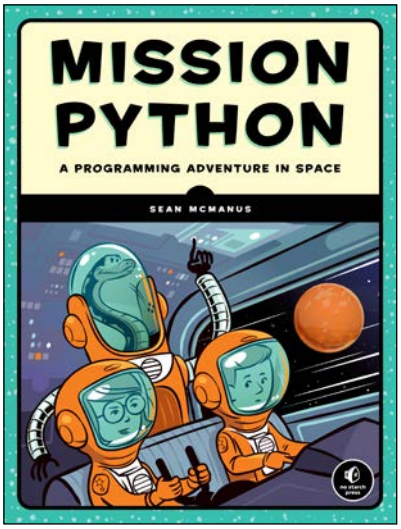
EVEN MORE BOOKS FOR FANS OF ALL AGES!

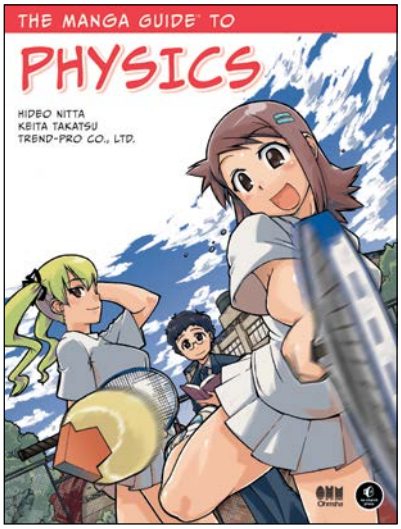


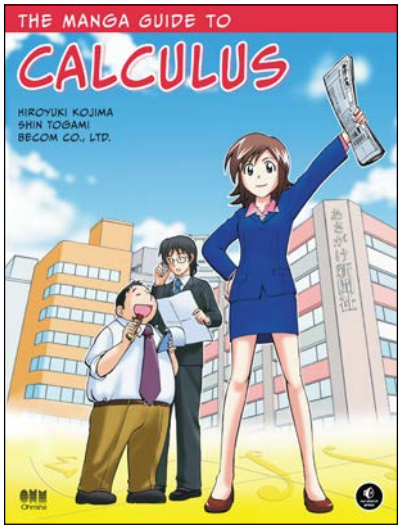


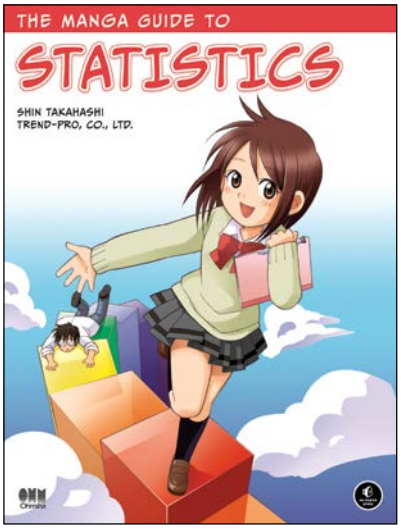












no starch press